The Interaction Network: a Performance Measurement and Evaluation Tool for Loosely-Coupled Distributed Systems

---

A thesis

submitted in partial fulfilment

of the requirements for the Degeee of

Doctor of Philosophy in Computer Science

in the

University of Canterbury

by

Paul Ashton

University of Canterbury

1992

PHYSICAL SCIENCES LIBRARY
THESIS
copy 2

Contents

Appendices

Figures

Tables

Abstract

Much of today's computing is done on loosely-coupled distributed systems. Performance issues for such systems usually involve interactive performance, that is, system responsiveness as perceived by the user. The goal of the work described in this thesis has been to develop and implement tools and techniques for the measurement and evaluation of interactive performance in loosely-coupled distributed systems.

The author has developed the concept of the *interaction network*, an acyclic directed graph designed to represent the processing performed by a distributed system in response to a user input. The definition of an interaction network is based on a general model of a loosely-coupled distributed system and a general model of user interactions. The author shows that his distributed system model is a valid abstraction for a wide range of present-day systems.

Performance monitors for traditional time-sharing systems reported performance information, such as overall resource utilisations and queue lengths, for the system as a whole. Performance problems are now much more difficult, because systems are much more complex. Recent monitors designed specifically for distributed systems have tended to present performance information for execution of a distributed program, for example the time spent in each of a program's procedures. In the work described in this thesis, performance information is reported for one or more *user interactions*, where a user interaction is defined to be a single user input and all of the processing performed by the system on receiving that input. A user interaction is seen as quite different from a program execution; a user interaction includes the partial or total execution of one or more programs, and a program execution performs work as part of one or more user interactions.

Several methods are then developed to show how performance information can be obtained from analysis of interaction networks. One valuable type of performance information is a decomposition of response time into times spent in each of some set of states, where each state might be defined in terms of the hardware and software resources used. Other performance information can be found from displays of interaction networks. The critical path through an interaction network is then defined as showing the set of activities such that at least one must be reduced in length if the response time of the interaction is to be reduced; the critical path is used in both response time decompositions and in displays of interaction networks.

It was thought essential to demonstrate that interaction networks could be recorded for a working operating system. INMON, a prototype monitor based on the interaction network

concept, has been constructed to operate in the SunOS environment. INMON consists of data collection and data analysis components. The data collection component, for example, involved the adding of 53 probes to the SunOS operating system kernel.

To record interaction networks, a high-resolution global timebase is needed. A clock synchronisation program has been written to provide INMON with such a timebase. It is suggested that the method incorporates a number of improvements over other clock synchronisation methods.

Several experiments have been performed to show that INMON can produce very detailed performance information for both individual user interactions and groups of user interactions, with user input being made through either character-based or graphical interfaces.

The main conclusion reached in this thesis is that representing the processing component of a user interaction in an interaction network is a very valuable way of approaching the problem of measuring interactive performance in a loosely-coupled distributed system. An interaction network contains a very detailed record of the execution of an interaction and, from this record, a great deal of performance (and other) information can be derived. Construction of INMON has demonstrated that interaction networks can be identified, recorded, and analysed.

Chapter 1

Introduction

This thesis is concerned with the measurement and evaluation of interactive performance for loosely-coupled distributed systems. In particular, the author introduces the concept of the *interaction network*, an acyclic directed graph designed to represent the processing done by a loosely-coupled distributed system in response to a user input. It will be shown that the interaction network is a potentially very powerful performance evaluation tool for measuring and evaluating interactive performance in loosely-coupled distributed systems.

A *loosely-coupled distributed system* consists of a collection of computer systems connected only by a communications network. For such systems, performance issues most commonly refer to interactive performance. Until the last decade, the main goal for the performance of a computer system was that the system throughput, that is the amount of useful work per unit time, should be maximised. This goal was important because maximum use had to be made of expensive hardware resources. Over time, however, as hardware prices dropped and systems became more interactive in nature, the performance goal of high system responsiveness has been progressively more important. The responsiveness goal can be expressed by saying that the time that a user has to wait for a request to be performed should be as small as possible. The goals of high throughput and of high responsiveness are not independent: for throughput to be high workload must be high, and responsiveness will decrease as workload increases.

In early work on performance, measures of responsiveness were defined for batch processing systems, where input was often submitted as a deck of punched cards, and the output received as a printout. Turnaround time was defined as "the delay between the presentation of input to a system and the receipt of output from it" [CALI67]. By 1967 the importance of responsiveness was increasing, as is suggested by the following quote from [CALI67]: "Of these measures [throughput, turnaround time, and availability], throughput is the one that has been used the longest, although the importance of turnaround time and availability is becoming increasingly clear to many users".

As the use of terminals increased, measures of responsiveness were defined for interactive systems. Perhaps the most widely used performance index is response time. Intuitively, response time is simply the time taken by the computer system to perform a

command specified by a user at a terminal, although many different definitions of response time have been given [PENN84].

Today, powerful workstations are common, and usually include a bit-mapped display and a pointing device, often a mouse. Graphical user interfaces (GUIs), have been designed and constructed for them. These workstations cost so little that "throughput" is no longer important. What is important is that the system responds quickly to user inputs. Users in interactive environments are very sensitive to delays, and GUI environments require more system resources than the older character-based interfaces.

In addition to the trend toward more sophisticated user interfaces, computer systems have become much more complex. Twenty years ago, a computer system consisted of a small number of major hardware components: a single main processor (CPU), main memory, peripheral devices, and sometimes additional processors to manage peripherals. Telecommunications networks were used only to connect remote peripherals, usually terminals and printers.

In the last two decades, two developments have contributed significantly to greater system complexity. First, systems with more than one main processor, often called multiprocessors or *tightly-coupled* distributed systems, have been constructed. Second, systems have been developed which consist of many computers connected only by a communications network. Such systems are called *loosely-coupled* distributed systems. In this thesis the term "distributed system", unless otherwise specified, means a loosely-coupled distributed system.

A system with a single main processor is a uniprocessor system, and a system that is not part of a distributed system is a centralised system. Systems with the early 1970s architecture described above were therefore uniprocessor, centralised systems. Each computer in a distributed system is a uniprocessor or a multiprocessor, and workstations are usually part of a distributed system.

The large number of components in a distributed system, and the complex ways in which they operate and interact, can make it difficult to determine the reasons for poor interactive performance. The interaction network is introduced in this thesis as a tool by which problems of poor performance can be solved.

The thesis is structured as follows:

In Chapter 2, a concise, general model is described of computation and communication in a distributed system. The model is shown to be one that represents a wide range of existing systems.

In Chapter 3, a review is made of computer performance evaluation (CPE). The areas of CPE most directly relevant to this thesis are discussed.

In Chapter 4, a simple model is introduced of the way in which a user interacts with a computer system. That model defines a *user interaction* to be a single user input to the computer system, and all of the processing performed by the system after receiving that input. Although the model is simple, it is also very general. As well, the interaction network is introduced, as a directed acyclic graph to represent all processing performed by the system as part of a single user interaction. The idea of the interaction network is based on the distributed system model from Chapter 2, and the user interaction model given at the beginning of Chapter 4.

By aiming to provide performance information for one or more user interactions the author has taken an approach substantially different from that of most other work on measurement of performance for distributed systems, in which performance information is provided for the execution of entire programs. A user interaction is quite different from a program execution, as a user interaction includes the partial or total execution of one or more programs, and a program execution performs work as part of one or more user interactions.

In Chapter 5, the critical path through an interaction network is defined. The critical path shows the series of activities performed during an interaction such that, if response time for the interaction is to be reduced, then the duration of at least one of the activities on the critical path must be reduced. Finding critical paths is important because, if system responsiveness is to be improved, then interaction response times must be reduced. A simple algorithm is presented for finding the critical path through an interaction network.

In Chapter 6, ways are considered in which performance information can be extracted from interaction networks. Methods are described for analysis of both individual interaction networks and sets of interaction networks. That is, the interaction network concept is valuable not just for analysis of single interactions, but also for sets of interactions such as all interactions that occurred on a particular day, or all interactions that included use of a particular node or set of nodes. In this way, analysis of interaction networks can lead to general improvements in interactive performance.

The ideas discussed to this point require some demonstration. In Chapter 7, issues for the design of performance monitors that record interaction networks are discussed. INMON, a prototype monitor constructed by the author to record interaction networks, is introduced.

A highly accurate global timebase must be available if reliable performance information is to be derived from interaction networks. In Chapter 8, a clock synchronisation program,

written to provide a global timebase for INMON, is described. Background to the clock synchronisation problem and results of validation experiments for the program developed are also given.

INMON consists of an interaction network recording component, described in detail in Chapter 9, and an interaction network analysis component, described in detail in Chapter 10. In Chapter 11, a number of experiments performed with INMON are described. The experiments show a range of situations in which INMON, and therefore methods based on interaction networks, can be applied.

Finally, in Chapter 12, a number of conclusions are drawn. Briefly, the main conclusions are that:

(1) The interaction network is potentially a very powerful tool for describing the processing performed by a loosely-coupled distributed system in response to a user input.

(2) Interaction networks can be used in performance analysis of individual interactions, and of sets of interactions. Where a set of interactions is to be analysed, membership of the set can be based on whether an interaction uses some object(s) of interest to the performance analyst, such as a particular node or other system resource.

(3) The decision to present performance information for interactions, rather than for programs, was sound. Presenting performance information for interactions is particularly appropriate where a graphical user interface is used.

(4) It is possible to design and construct monitors based on the interaction network concept. The author has demonstrated, by modifying a version of SunOS, that interaction networks can be recorded for a working operating system. While these modifications were not trivial, installation of software to record interaction networks should be much easier if considered at the time of operating system design.

(5) Because interaction networks contain a great deal of information on the behaviour of software systems, they can be valuable in other areas, such as in teaching about operating systems, and in program debugging.

Some of the ideas explored in this thesis are a continuation of earlier work on decomposing response times into components [ASHT84], [PENN84], [PENN86], [PENN88]. Some of the material in this thesis has already been published. Much of Chapters 2 and 4, and some of Chapters 5 and 6, are covered in [ASHT91a], and Chapter 8 is summarised in [ASHT92]. Also two technical reports have been published, with [ASHT91b] an expanded version of [ASHT91a], and [ASHT90] containing early drafts of Chapters 2 and 4.

Chapter 2

A Model for Computation in Distributed Systems

Before the performance of a system can be measured, it is necessary to have an understanding of the nature of that system. In this chapter, we give a definition for the term *distributed system* by (1) quoting a widely accepted definition for the hardware components of a distributed system, and (2) introducing the author's communicating threads model, a very general model for the way in which computation and communication are supported by operating systems for distributed systems. The interaction network approach to measurement of interactive performance in distributed systems is based on the definition of a distributed system given in this chapter.

The hardware definition and the communicating threads model are described in Section 2.1. In Sections 2.2 and 2.3, it is shown that the model applies to the classes of operating system commonly found in distributed systems. Finally, comparisons with other models are made in Section 2.4, and the chapter is summarised in Section 2.5.

2.1 Distributed System Definition

A *(loosely-coupled) distributed system* is defined for the purposes of this thesis as:

> A collection of separate computer systems (*nodes*) that communicate and synchronise only by passing messages across a communication network .

In this thesis, the term distributed system is used always to refer to loosely-coupled systems. In the literature, the term distributed system is often used as a synonym for distributed operating system. This is not our usage, as will become apparent in the remainder of this chapter.

As all inter-node communication in a distributed system is through a communication network, nodes cannot share memory, clock signals, or devices. A node can be any sort of computer system: a general-purpose computer system (either a uniprocessor or a multiprocessor system), possibly with attached devices; or a special-purpose system such as a file server, or a print server. The communication network is typically a local area network (LAN), or an internet of LANs connected together by wide area network (WAN) links. Common examples of distributed systems are PC and workstation LANs.

This definition of the hardware components of a distributed system is widely used (see for example [COUL88], [GOSC91], [SILB91]), and its use will not be justified further.

The two fundamental activities in a computer system, distributed or centralised, are *computation* and *communication*, where the latter includes both exchange of data and synchronisation of execution. The *communicating threads* model that we now introduce describes how operating systems for distributed systems support computation and communication.

In this model, computation is performed by processors following *threads* of execution, each thread being "a schedulable unit of flow of control" [BRAN89]. Threads are the units of CPU scheduling, and can be created and terminated dynamically.

Two threads can communicate either through an area shared memory, or by message passing [ANDR83]. It follows from the definition of the hardware components of a distributed system that shared memory can be used for intra-node communication only. Message passing must be used for inter-node communication, and is often used for intra-node communication as well. Systems exist that provide programmers with a shared memory abstraction for threads executing on different nodes [BAL89] [TAM90], but these systems use message passing to provide this abstraction.

Message passing allows threads to exchange data, as each message has associated data, and to synchronise their activities, as a message cannot be received before it has been sent. Areas of shared memory allow threads to exchange data, by writing values to and reading values from shared variables, but they do not directly support synchronisation. Several process synchronisation mechanisms have been developed, all of which make use of shared variables [ANDR83]. In the following, "communicating using shared memory" is taken to include both data transfer using shared variables, and synchronisation using these synchronisation mechanisms.

The *communicating threads* model can now be summarised. Computation in a distributed system is performed by threads executing on nodes. Threads communicate by passing messages, and perhaps through shared memory.

The next section contains more detailed discussion of operating system support for threads, message passing, and shared memory.

2.2 Operating Systems for Distributed Systems

Several different types of operating systems have been constructed for use in distributed systems. Two widely used classifications for these operating systems are introduced in Subsection 2.2.1 and one, classification of systems into process-oriented

systems and object-oriented systems, is used in the remainder of this chapter. We then describe the forms in which threads (Subsection 2.2.2), communication by message passing (Subsection 2.2.3), and communication using shared memory (Subsection 2.2.4) are provided in process-oriented systems and in object-oriented systems. From these descriptions, it is clear that the communicating threads model describes well computation and communication in both process-oriented systems and object-oriented systems.

In Section 2.3, many examples are given of process-oriented and object-oriented systems that have been constructed. These examples substantiate descriptions given in this section of the characteristics of process-oriented and object-oriented systems.

2.2.1 Classifications

Fortier [FORT88] has identified three phases in the evolution of operating systems for distributed systems:

- early communication services
- network operating systems
- distributed operating systems

Operating systems with early communication services provided users with logical machine names (rather than physical machine names), reliable communications, and services such as remote login, and file transfer.

Network operating systems evolved from operating systems with early communication services. They use the same centralised operating systems that were used to provide early communication services, but have an extra component, the network operating system layer, which makes the presence of the communication network transparent to users and to the local operating system.

Distributed operating systems also provide network transparency, but have global, network-wide, policies for resource management. Network operating systems, on the other hand, have local policies for resource management because they are created by extending local operating systems.

Tanenbaum and van Renesse [TANE85] also talk about network operating systems and distributed operating systems, but give somewhat different definitions. They define: a network operating system to be one in which the machine boundaries are visible to the user; and a distributed operating system to be one in which the machine boundaries are transparent, that is the user is presented with a virtual uniprocessor.

The distinctions between the classes introduced above are based on the services provided, the network transparency, and the resource management policies. These

considerations are however above the level at which we wish to model operating systems for distributed systems. Fortier also classifies operating systems as being either process-oriented or object-oriented, and this classification is at a level comparable to that of the communicating threads model.

The main elements of a process-oriented system are processes and messages. All actions are performed by processes, and processes pass messages when they need to communicate with other processes. The basic structure in an object-oriented system is the object. An object is self contained, containing both private data and a set of methods. Methods can be invoked from outside an object, and an object's private data is accessible only to the methods of the object. Most existing systems fit well into one or other of these categories.

The remaining subsections of this section show that the communicating threads model describes both process-oriented and object-oriented systems.

2.2.2 Threads

We now describe the ways in which threads are used to provide computation in process-oriented systems and in object-oriented systems.

(1) Process-oriented systems

The term process is usually taken to mean a single thread of execution within a dedicated address space. Processes in Unix, for example, conform to this definition [RITC74]. The execution context of such a process is "heavyweight", because there is considerable overhead involved in creating and destroying the execution context of a process, and in context switches, where a processor switches from executing one process to executing another. The address space is one item of process context that is particularly "heavy" in terms of the overhead involved in manipulating it. Other common items of process context include the execution history of the process (usually a run-time execution stack), saved processor state, open objects (including files and communication ports), scheduling information, resource usage records, and resource usage constraints [SILB91].

Many recent operating systems permit several *threads* of execution to share an address space, reducing the overhead of creating and destroying threads within an existing address space, and of switching between threads in the same address space. A *cluster* is an address space and other items of context shared by one or more threads [MULL86], [COUL88]. Each thread has at least a stack and the saved processor state as items of context. Because all threads in a cluster share the same address space, and because nodes do not share memory, a cluster and all its threads must be located on the same node at any given time. A

process can be regarded as a special case of a cluster, that is a cluster that contains only one thread.

Two types of thread implementations exist [BERS91]:

(i) *Kernel-level threads*, where support for thread creation, termination, and scheduling, is provided within the operating system kernel.

(ii) *User-level threads*, where thread support is provided outside the kernel by a library of functions linked in with each application.

Usually, clustering of threads introduces a two-level process structure, with user- or kernel-level threads at the lower level, and clusters of threads at the higher level. A three-level structure is possible, however, where one or more user-level threads can execute within the context of each kernel-level thread, and one or more kernel-level threads share the address space provided by a cluster [BERS91].

Many applications are performed by several communicating clusters, and it is sometimes desirable to be able to deal with all of the clusters involved in a particular application as a group. This can be done by adding a further level to the process hierarchy. A *process group* is one or more clusters that are related in some way, and which are to be treated as a single entity for some reason. Clusters within a process group may be on different nodes. The term cluster group may seem more appropriate, but use of the term process group is widespread in the literature.

In most process-oriented systems then, there is a one to four level hierarchy of process constructs, with all computation performed by processors following threads of execution within that hierarchy. Often there is a single level hierarchy consisting solely of processes. At the other extreme, a system might provide user-level threads, which execute within the context of kernel-level threads, which execute within clusters, which are grouped into process groups. For any particular system, the "threads" referred to by the author's communicating threads model are at the lowest level of the process hierarchy of that system.

(2) Object-oriented systems

In object-oriented systems, threads are supported in either of two ways. The first possibility is that all objects are *passive*, that is, they have no associated threads. Threads are structured into processes and/or clusters and/or process groups in the same way that they are in process-oriented systems.

An alternative approach is for some or all objects to be *active*. Active objects have thread(s) associated with them and provide most items of context for each of those threads. Methods of passive objects are usually performed by the invoking thread for local

invocations, and agent threads in remote invocations. Methods of active objects may be performed in the same way that passive object invocations are performed, or they may be performed by one of the threads active within the invoked object.

In some object-oriented systems two or more threads can execute concurrently within an object. This is the equivalent of two or more threads executing concurrently or in parallel within a cluster in the process-oriented model.

(3) Thread summary

Operating systems provide support for threads in several different ways. In process-oriented systems, a process hierarchy of up to four levels can exist, from user-level threads right up to process groups. Object-oriented systems may use similar structures, or may associate threads with active objects. The communicating threads model holds for all of these different structures, as in all of them the work is done by threads, with only the environment within which the threads execute differing between the different systems.

In the discussion of threads to this point, the topic of thread migration has not been discussed. In some distributed systems, threads can migrate, that is a thread can be suspended part way through its execution, transferred to another node, and then resume its execution. Clearly, a thread cannot migrate to another node independently of its address space, so clusters and active objects are the usual units of migration. It may seem that the communicating threads model should make some provision for thread migration, but this is not the case. When, say, a cluster migrates, it is simply sent as one or more messages between threads that implement migration. As message passing is included in the communicating threads model, no extra constructs are required in the model to allow for thread migration.

2.2.3 Message Passing

Passing a message involves one thread sending a message to another thread. To send a message some sort of address is required to identify the destination. Message passing can be used for both inter-node and intra-node communication.

Communication using message passing occurs in both process-oriented and object-oriented systems. Message passing is explicit in process-oriented systems, and may be direct or indirect. With direct message passing, the sending thread specifies the thread it wishes to send the message to, and the message is sent directly to that thread. In indirect message passing, the sending thread sends the message to a message buffer called a *mailbox*. The message remains at the mailbox until a receiving thread requests a message from the mailbox. The message travels indirectly, via the mailbox, from the sending thread

to the receiving thread [ANDR83]. Broadcast and/or multi-cast send operations, which send a single message to several destinations, may also be available.

Message passing is implicit in object-oriented systems. All invocations of (methods in an) object can be regarded as requiring a pair of messages: a message from the invoking object to the invoked object when an object is invoked, and a reply message from the invoked object back to the invoking object when the invocation is complete. This message protocol passing is a type of remote procedure call protocol, which is discussed further below. In practice, all inter-node and some intra-node invocations are done by passing messages. Two threads are involved: a thread executing in the invoking object, and a thread that executes the invoked method in the invoked object. Most intra-node invocations are performed in a way very similar to that of local procedure calls, in that the thread that executes an invocation is also the thread that executes the invoked method.

For two threads to communicate using message passing, the sending thread specifies the destination of the message and the message contents, and sends the message using a send operation. The receiving thread invokes a receive operation to receive the message. If message passing is indirect, then the receiving thread must provide the receive operation with the name of a mailbox.

Each of the send and receive operations may be blocking or non-blocking. A blocking send blocks the sending thread until the receiving thread receives the message. A non-blocking send delays the sending thread only for long enough to put the message into a buffer. A blocking receive blocks the receiving thread until a message is available. A non-blocking receive checks to see if a message is available; if a message is available it is received, otherwise the non-blocking receive returns an indication that no message is currently available. The blocking send and receive operations together provide *synchronous* message passing, that is the sender and receiver synchronise with each other when the message is passed. The non-blocking send and blocking receive operations together provide *asynchronous* message passing [BAL89].

Higher level protocols can be constructed from the send and receive operations. *Remote procedure call* (RPC) is a message passing protocol that is widely used in distributed systems [BIRR84], [COUL88]. The idea behind RPC is that a form of ordinary procedure call extended to provide interprocess communication should be easy for programmers to understand and use. RPC protocols are asymmetric, in that one thread is the caller of the remote procedure, and the other thread executes the remote procedure. These threads are often referred to as the client thread and the server thread.

When a client calls a remote procedure the arguments of the remote procedure are packed into a message which is sent to a server thread. The server receives the call message, executes the requested procedure using the arguments in the message, and packs

the return values into a message which is sent back to the client. The client receives the return message and retrieves the return values of the remote procedure. Remote procedure protocols are usually *synchronous*, where the client thread blocks until the return message is received, but some are *asynchronous*, where the client thread can continue executing while the remote procedure is executing.

In many distributed systems, each system service is implemented by one or more server threads which accept remote procedure calls from clients, with common services including file, directory, and authentication services. These systems are said to be based on the *client/server model*. Each service provides a set of procedures (the service interface) which users of the service (client threads) can invoke using remote procedure calls. A file service, for example, will usually include procedures for creating files, reading files, writing files, and deleting files.

2.2.4 Shared Memory

Communication using shared memory is possible if the communicating threads can access a common set of variables, that is the shared variables are in an area of memory that is part of the address space of each thread. Communication occurs when one thread writes data to a shared variable, and this data is subsequently read by other threads. Shared memory communication can only occur within a node, as by definition nodes do not share memory.

When threads communicate using shared memory they can pass data by reading and writing shared variables, but synchronisation is not provided. Separate synchronisation mechanisms must therefore be provided. All of these mechanisms, including Dekker's algorithm, monitors, and semaphores, need to use shared variables [DEIT84]. Synchronisation mechanisms are usually required to manage access to shared variables so that concurrent update problems are avoided.

Many operating systems share memory between address spaces to ensure that physical memory is used efficiently. Examples of this type of memory sharing include sharing of re-entrant code, and copy-on-write techniques. These types of memory sharing are not for communication purposes, and so are not considered further.

In process-oriented systems and some object-oriented systems, shared memory communication can be performed by threads within the same cluster, and by threads in different clusters if the shared variables used for communication are common to the address spaces of both clusters. In other object-oriented systems, threads that wish to communicate using shared memory can do so only through the local data of an object whose methods can be performed by both communicating threads. Synchronisation mechanisms such as semaphores and monitors are implemented in much the same way in both types of system.

2.3 Existing Systems

Several process-oriented and object-oriented systems are now briefly described. These examples show that many existing systems are consistent with the general descriptions of process-oriented and object-oriented systems given in Section 2.2. Because of this consistency, all of the systems discussed can be described by the communicating threads model, as it was shown in Section 2.2 that the model applies to the general descriptions of both types of system.

2.3.1 Process-oriented Systems

The five process-oriented systems described can be classified as :

- Centralised operating system with early communication services: Berkeley Unix.
- Network operating system: SunOS.
- Distributed operating system: Amoeba, Mach, V.

(1) Berkeley Unix 4.3

Berkeley Unix (BSD) is an operating system for a centralised uniprocessor system [LEFF88]. Computation is performed by *processes*, with each process containing a single thread of execution. Each process is a member of one *process group*. The command interpreter assigns processes to process groups, and provides user commands to allow all processes in a progress group to be started, suspended, restarted, and terminated.

The main form of message passing is indirect, with *messages* sent to *sockets*. Also, processes can pass very simple messages using *signals*, a direct form of message passing. Processes cannot communicate using shared memory.

(2) SunOS

SunOS is a Unix-based network operating system, originally derived from 4.2 BSD. Computation is performed by *processes*. A library is provided that supports user-level threads known as *lightweight processes* [SUN88a], but it seems to be seldom used.

SunOS provides a variety of mechanisms for interprocess communication. Message passing is provided by sockets and signals (from Berkeley Unix), and by *message queues*, an indirect form of message passing introduced from Unix System V [BACH86]. SunOS also includes *Sun RPC*, a remote procedure call implementation layered on top of sockets. Sun RPC is used in the SunOS network services, all of which are based on the client/server model [SUN88b]. SunOS also provides shared memory communication. Processes can

share areas of their address spaces using the Unix System V *shared memory* mechanism, and can synchronise using Unix System V *semaphores* [BACH86].

(3) Amoeba

Amoeba is a distributed operating system developed at the Vrjie University [MULL86]. Computation is performed by kernel-level threads called *tasks*, with one or more tasks executing in the address space of a *cluster*.

Message passing is indirect, with *messages* sent to *ports*. Amoeba's message passing protocol is very similar to RPC in that it involves <request, reply> message pairs. Tasks within a cluster can communicate via shared memory, but tasks in different clusters cannot. In Amoeba 3.0 [TANE89] no explicit synchronisation mechanisms are provided. Tasks within a cluster are not pre-emptable and run until they are logically blocked, at which point another task in the cluster can begin executing. Protection of shared data structures relies on the data structures being in a consistent state wherever a thread can logically block. This method was found to be unsatisfactory and, in Amoeba 4.0, semaphore and mutual exclusion primitives are provided, and threads are no longer guaranteed to run until they logically block.

Tanenbaum *et al* [TANE89] describe Amoeba as an "object-based" system. Each server manages a collection of objects with, for example, the file server managing a collection of file objects. Each server, then, is an object manager, and is willing to perform a set of operations on the objects it manages. When a client makes a remote procedure call to a server, it provides as parameters: a capability, the operation it wants the server to perform, and parameters for the operation. The capability contains the port number of the server, an object number, and a rights field that specifies the operations that the capability entitles the capability holder to perform on that object. Thus Amoeba is implemented in a process-oriented way, but operates in an object-based fashion.

(4) Mach

Mach is a distributed operating system developed at Carnegie-Mellon University [MASO87]. Computation is performed by kernel-level threads, with one or more *threads* executing in the address space of a *task*. A task provides the threads it contains with protected access to resources such as an address space, and capabilities for ports (that is, most items of context are associated with tasks). Tevanian and Smith comment that the Unix process abstraction is simulated in Mach by combining a task and a single thread [TEVA89].

Message passing is indirect, with *messages* sent to *ports*. Threads in the same task can communicate via their shared address space, and tasks on the same node may share areas of memory that contain shared variables.

(5) The V kernel

The V kernel is a distributed operating system developed at Stanford University [CHER84]. Computation is performed by kernel-level threads called *processes*, with one or more processes executing in the address space of a *team*. When a process is created it is created either as a member of the team of the creating process, or as the first (root) process of a new team. Each process can be a member of one or more *process groups* [CHER85]. Operations that can be performed on a process group include: sending a message to a process group (a message is sent to all processes in the group), receiving a message from a process group (a message is received from any process in the group), killing all processes in a group, and sending a signal to all processes in a group.

Message passing is direct, with *messages* sent to processes. The message passing protocol includes primitives to support RPC, and communication is usually done in an RPC-like way. Processes in the same team can communicate using shared memory, but processes in different teams cannot.

2.3.2 Object-oriented Systems

Three object-oriented systems are introduced. Of these, Clouds and Eden are best classified as distributed operating systems, while Emerald is an object-oriented language and run-time system for distributed systems.

(1) Clouds

Clouds is an object-oriented distributed operating system project at Georgia Tech [WILK89]. In Clouds, *threads* perform computation, and all objects are passive. When a thread is created it starts executing at the entry point of some object. The address space of a thread consists of the object it is currently executing in plus some private stack space. When a thread invokes a local object, the object component of the address space of the thread is switched from the invoking object to the invoked object, and the thread executes the invoked method itself. When the invocation is completed, the object component of the address space of the thread is switched back to the invoking object. For invocations of remote objects a remote procedure call mechanism is used. It is possible for several threads to execute within an object concurrently, or in parallel on a multiprocessor node.

(2) Eden

Eden is an object-based distributed operating system developed at the University of Washington [ALME85]. The main construct in Eden is the Eden object, or *Eject*. An Eject may be active or inactive. In Eden, threads are known as *processes*, and each process is associated with an active Eject. An active Eject has an address space and at least one process executing within it, and an inactive Eject is dormant and is saved on disc. Every Eject invocation is handled by a process local to the invoked Eject. For each invocation a process in the invoking Eject sends an invocation message, a process in the invoked Eject receives the invocation message, performs the invoked method, and sends a reply message to the invoking process. Finally, the invoking process receives the reply message. Messages are used in both local and remote invocations.

Multiple processes may be active in a single Eject, and are synchronised using *monitors*.

(3) Emerald

Emerald is a "distributed object-based language and system" developed at the University of Washington [JUL88]. In Emerald, threads are known as *processes*, and objects can be active or passive. Every process in Emerald is associated with an active object. Object invocation is performed by the calling process. In remote invocations the calling process migrates to the node where the invoked object is located, performs the invoked method, then migrates back to the node where the invoking object is located. Migration of a process from a source node to a destination node requires messages to be sent between threads on the source and destination nodes.

Multiple processes may be active in a single object, and are synchronised using *monitors*.

2.4 Other Models

Most approaches to the measurement of the performance of distributed systems are based on some model for how computation is performed in a distributed system. Models used in four approaches are briefly presented, and are then compared with the author's communicating threads model.

2.4.1 DPM

In the model of distributed computation used by DPM [MILL86], a distributed system executes *distributed programs*. Each distributed program is a collection of *processes* co-operating on a common task. Processes execute on *machines*, and the processes that make

up a program may execute on different machines. A process is an address space that contains a single thread of execution, and processes cannot communicate using shared memory. Communication between processes is based on messages.

This model is similar to the communicating threads model, but is much less general as, for example, processes cannot share memory, and each address space contains a single thread of execution.

2.4.2 IPS

In the model of distributed computation used by IPS [YANG89], a distributed system executes *programs*. Components within a program are modelled in a hierarchical fashion. Work for each program is performed on one or more *machines*; on each of these machines reside one or more of the program's *processes*; internally, each process is implemented using *procedures*; within each procedure *primitive activities* are performed. Communication between process is by message passing.

This model is similar to the communicating threads model, but again it is less general. Yang and Miller comment, however that the hierarchy is not fixed, and that new levels can be added as required by the environment being modelled [YANG89].

2.4.3 TMP

In the model of distributed computation used by TMP [WYBR88], a distributed system executes *programs*. Components within a program are modelled in a hierarchical fashion, in a similar way to that of IPS. Work for each program is performed by on one or more *machines*; on each of these machines reside one or more *distribution units*; each distribution unit is one or more *processes* that communicate using shared memory. Processes in different distribution units cannot communicate using shared memory, so all such communication is done using synchronous or asynchrounous message passing.

This model is similar to the communicating threads model, but again it is less general.

2.4.4 Relational Approach

The model used in the relational approach described by Snodgrass [SNOD88] is a very general one, intended to be applicable to all "complex systems", with distributed systems considered to be one type of complex system. In that model, a system consists of a collection of *typed entities*, with each entity type managed by a *type manger* which exports a set of *operations* that can be performed on instances of that type. Data on system operation is collected by *sensors*, each of which is placed in a type manager and is associated with one of the operations exported by the type manager.

2.4.5 Summary of Other Models

In the first three models described, computation is performed by distributed programs each of which consists of one or more communicating processes, and each process performs work as part of one and only one program. The communicating threads model is sufficiently general to apply to all systems described by these models. Most distributed monitoring systems (that is, systems for the performance monitoring of distributed systems), have the (distributed) program as their unit of analysis. In the work described in this thesis, however, the interaction is the unit of analysis, as will be explained in Chapter 4.

Snodgrass' relational approach has a model that is more general than the communicating threads model, as it is intended to be capable of describing all "complex systems", with distributed systems considered to be one type of complex system. As the communicating threads model describes a very wide range of distributed systems, we see no advantage in making our model as general as Snodgrass'.

2.5 Summary

The hardware structure of a distributed system has been defined, and a model has been developed to represent the way in which computation and communication are supported by operating systems for distributed systems. The hardware definition and the communicating threads model are stated briefly as:

> A distributed system consists of a number of (uniprocessor and/or multiprocessor) nodes communicating only via a communication network. Processors in each node execute threads. Threads communicate and synchronise using message passing and perhaps shared memory.

The model is concise, and general. The model describes very well a wide variety of operating systems for distributed systems, as has been shown by comparing the model with general classes of operating systems for distributed systems, with specific examples of operating systems for distributed systems, and with models of distributed systems used in performance monitors for distributed systems.

The model forms a major part of the basis of the interaction network concept, an approach to measuring the interactive performance of distributed systems that is to be described in the remainder of this thesis.

Chapter 3

Performance Background

Computer performance evaluation (CPE) is a broad field. This chapter gives a brief overview of the field, then focuses on the areas of CPE relevant to the work described in this thesis. In CPE, *performance data*, data that in some way characterises the performance of a (real or modelled) computer system, is gathered and analysed to produce *performance information*. Performance information is used in *performance evaluation studies* that are conducted for a wide variety of reasons. For example, a study might be carried out to determine reasons for poor interactive performance of an application or system, or to determine the relative performance of a number of systems during a procurement exercise. The person conducting a study is known as a *performance analyst*.

Classifications of performance evaluation studies are given in Section 3.1, and the types of studies that involve measurement of interactive performance for distributed systems are identified. Overviews of performance data and performance information are given in Sections 3.2 and 3.3. Types of performance information that have proved to be useful in performance analysis of interactive systems and distributed systems are described in Sections 3.4 and 3.5. The chapter is summarised in Section 3.6.

3.1 Performance Evaluation Studies

Performance evaluation studies can be categorised in a number of fundamental ways:

(1) The evaluation techniques used in the study. Two types of evaluation techniques, *modelling* and *measurement*, are used to gather performance data.

(2) The way in which performance data is used in the study. The primary uses of performance information are in *comparisons* of the performance of two or more systems, and in *diagnosis* of the reasons for poor performance.

(3) The viewpoint of the performance analyst who conducts the study. A system may be seen differently by different groups associated with the system. These groups include hardware and system software designers, installation managers, analysts and programmers, and end-users. Each group has different views on which aspects of computer performance are important.

(4) The reason for the study. Most performance studies are conducted as part of one of the following activities: procurement of new systems or subsystems, improvement of existing systems, capacity planning for future systems' requirements, and design of new systems [FERR83].

Categories (1), (2), and (3) are discussed further in Subsections 3.1.1 to 3.1.3. The types of performance study for which the interaction network concept is intended are described in Subsection 3.1.4.

3.1.1 Evaluation Techniques

Performance evaluation techniques are often classified as either *measurement* techniques or *modelling* techniques (see for example [FERR83]). Measurement techniques are designed to gather performance data from direct observation of computer systems. A system to be measured must therefore exist, and be available for measurement.

Modelling techniques are designed to gather performance data by evaluating models of computer systems. There are two types of models: simulation models and analytic models. A simulation model reproduces in (simulated) time certain aspects of the dynamic behaviour of a system [FERR83]. Recording performance data during a simulation has some similarity to measuring a real system, as the simulation model reproduces aspects of the behaviour of a real system. These models are usually implemented by writing programs that simulate the behaviour of systems.

Analytic methods use mathematical models to describe systems. These models are solved to produce performance information. The need for a model to be mathematically solvable limits the degree of detail with which an analytic model can represent a system.

Modelling techniques can be used to study systems that are not yet constructed, and can also be used to study many different system configurations in a much shorter time than it would take to actually set up and measure all of the different configurations. However, the accuracy of models in predicting performance of the systems that they model is limited because simplifying assumptions must be made in producing a model of a system, and in modelling system workload.

3.1.2 Uses of Performance Data: Comparative and Diagnostic

Performance data has two major uses: comparative and diagnostic. Many different kinds of comparisons can be made. In procurement, the performance of different systems can be compared. In improvement studies, the performance of various configurations of hardware and software can be compared to determine the best configuration. Also, actual system performance can be compared against desired system performance. In capacity

planning, the performance of various options for future systems is compared. In hardware and software design, the performance of different designs is compared.

Performance data is used for diagnostic purposes in improvement studies. In such studies, performance data is used to pin-point components of a system that are causing performance problems. Therapy to solve the performance problem can then be performed. Improvement studies can be categorised as a cycle of: *diagnose* performance problem, apply *therapy* to fix the problem, *compare* the performance after therapy with the performance before therapy to assess the effectiveness of the therapy.

3.1.3 The Study Viewpoint

Ferrari *et al* [FERR83] noted that there are many groups for whom the evaluation of a system's performance is of interest. They described a three level model of groups interested in performance evaluation: system designers; installation managers; and analysts, programmers and other users. System designers take a very broad view, as they must ensure adequate performance over the range of applications for which a system may be used. Installation managers have a narrower view, as they are interested only in providing adequate performance for the systems in their own installation. Individual users have a very narrow view, as they are interested only in receiving adequate performance for their own work.

While all groups are interested in the evaluation of a system's performance, different groups have different performance priorities. System designers may take a very system-oriented view of performance, aiming, for example, for a balanced utilisation of system components. Individual users, on the other hand, are more interested in how responsive the system is to their requests. The responsiveness of a system to interactive users is also known as *interactive performance*. Installation managers try to balance the use of system components, and in the process to provide interactive users as a group with good interactive performance.

3.1.4 Summary

The broad goal of the work described in this thesis, is to develop techniques for *measuring* the *interactive performance* of *distributed computer systems*. Therefore, this work is relevant for the following:

(1) Measurement techniques. Also, methods for analysis of data gathered by measurement could be useful in analysis of data gathered from simulations.

(2) Comparative and diagnostic studies. In a comparative study, we might compare the levels of interactive service provided by different systems (useful in procurement), or might

compare the levels of interactive service provided by different configurations of the same system (useful in improvement). Diagnostic studies are intended to uncover reasons for poor interactive service, either for an individual user, or for the user population as a whole.

(3) The viewpoints of, primarily, the user and the installation manager. Individual users are interested in the interactive performance for their requests. Installation managers are interested in the interactive performance of the requests of the entire user population, usually expressed as an average of some sort.

(4) Improvement studies and, to a lesser extent, procurement studies.

3.2 Performance Data

To measure the performance of a system, *performance data* must be collected. Techniques for collecting performance data are described in Subsection 3.2.1, and problems that arise in collecting performance data for distributed systems are discussed in Subsection 3.2.2.

3.2.1 Data Collection Techniques

Performance data is collected using two basic techniques: event detection and state sampling. Both techniques are based on the following model of a system. A system is assumed to take a finite number of *states*. Each change in state is caused by the occurrence of an *event*. The behaviour of a system can be represented by its progression through various states over time. In data collection based on event detection, information about each occurrence of one or more types of event is recorded as performance data. In data collection based on state sampling, selected parts of the system state are recorded at regular intervals.

There are advantages and disadvantages associated with both techniques. Event detection provides a very accurate description of system behaviour, but if events occur frequently the overhead imposed by event detection can be unacceptably high. Sampling is a statistical technique, so many samples may be required to achieve acceptable accuracy. Also, it may not be possible to sample often enough to detect events whose effects on system state are short-lived. However sampling overhead can be controlled by altering the sampling frequency. Monitoring overhead can be reduced to some desired level at the cost of extending the time required to collect a given number of samples.

Because of these advantages and disadvantages, the selection of a data collection technique depends on several factors. In some situations, one of the techniques may be ruled out because it cannot be implemented. In other situations, there is a trade off between the ability of the event detection technique to capture all events and associated data, and the

lower overhead of the sampling technique. Sometimes a mixture of event detection and sampling techniques is appropriate.

3.2.2 Problems with Distributed Systems

Problems that arise in monitoring of centralised systems have been extensively studied. To monitor distributed systems, some additional problems must be solved:

(1) There is no global timebase, as each node in a distributed system has its own clock. Algorithms have been developed to keep independent clocks synchronised to within a certain tolerance, but it is not possible to get an absolute ordering of events that occur in a distributed system [LAMP78].

(2) It is impossible to take an instantaneous snapshot (a sample) of the state of a distributed system. A number of snapshots must be taken, and these snapshots put together to describe the overall state of the distributed system. Because it is impossible to synchronise exactly the times at which the individual snapshots are taken, the overall state is described by pieces of state that existed at different times.

(3) The number of components in the system (nodes and network links between nodes), and the number of states that the system can take, are obviously much greater than is the case in centralised systems.

3.3 Performance Information

Raw performance data is collected by sampling and/or event detection. This raw data is at a very primitive level, and is usually processed into performance information before it is presented to the performance analyst. Performance indices provide performance information at an overview level, and they are described in Subsection 3.3.1. Other types of performance information are discussed in Subsection 3.3.2.

3.3.1 Performance Indices

Ferrari [FERR78] defines a *performance index* to be a descriptor that is used to represent a system's performance, or some of its aspects. An individual reading of a performance index is a single numeric value. Distributions of the values of a performance index over time, and statistics of distributions (particularly the mean and standard deviation), are of interest to the performance analyst. Graphs of performance indices against variables such as time and system workload are also of interest.

Ferrari identifies three main classes of performance index: productivity indices, responsiveness indices, and utilisation indices. *Productivity* indices show how much work the computer system is processing, and include: throughput rate, and capacity (maximum

throughput rate). *Responsiveness* indices measure how well a computer system is responding to requests from users, and include: response time, turnaround time, resource usage times, queueing times, and response ratio. *Utilisation* indices show the proportion of time that a hardware or software system component is busy, and include: processor utilisation, IO device utilisation, and operating system module utilisation. Some performance indices do not fit into any of Ferrari's categories; the number of page faults per second is an example.

Because index values are usually expressed as a single value (which may be a mean), or as a function of time or system workload, it is easy to compare values of performance indices measured in different sessions. Such comparisons should only be made if the values have been calculated in the same way, and particular care is required when comparing index values measured on different systems. One problem is that there is no widely agreed definition for some performance indices, including the very important index response time [PENN84]. Indices may be of little use in diagnostic studies because the values of indices by themselves do not provide enough information to determine the causes of poor performance.

3.3.2 Other Types of Performance Information

Many types of performance information cannot be expressed as performance indices, in particular the detailed information required for diagnostic studies. Often, these more detailed types of performance information provide information on the resource usage of various users and/or programs and/or program modules, so that both system bottlenecks and heavy resource users can be identified. Examples of these types of performance information are given in Sections 3.4 and 3.5.

3.4 Interactive Systems

The author's main interest lies in the interactive performance of distributed systems. Performance information relevant to the evaluation of interactive performance is discussed in this section, and performance information relevant to the investigation of the performance of distributed systems is considered in the following section. The techniques to be discussed in this section were all developed originally for use on centralised systems.

A description of how a user interacts with an interactive computer system is given in Subsection 3.4.1. The major types of performance information used in comparative performance studies are described in Subsection 3.4.2, and the major types of performance information used in diagnostic performance studies are discussed in Subsection 3.4.3. Finally, in subsection 3.4.4, a description is given of the types of performance information provided by a monitor developed by the author in earlier work.

3.4.1 A Description of User Interaction

Interactive performance is performance as perceived by a user who is interacting directly with a computer system. Traditionally, interactive users communicated with a system using character-oriented terminals. Input was typed on the terminal's keyboard and output was displayed as characters on the terminal's display. This type of terminal provides a *character-oriented user interface*, as both input and output consist of character streams. In the last decade, high-resolution bit-map displays and pointing devices, such as mice, have become readily available. These advances have made possible terminals that support *graphical user interfaces*, where typically output consists of raster images, and input consists of characters, mouse movements, and mouse button state changes. Sound output is becoming common, and possibilities for voice input are being researched.

A user conducts a dialogue with an interactive system, usually through a character-oriented terminal or a graphics terminal. A dialogue consists of one or more *interactions*. The author will define an interaction as a request from the user, and the processing (which may include output) that results from the request. In the next chapter, interactions are discussed in much more detail.

3.4.2 Comparative Studies

Performance indices from Ferrari's responsiveness class of indices [FERR78] are usually used in comparative studies of interactive performance. By far the most widely used of these is response time, discussed in (1) below. Response ratio, another index of responsiveness, is described in (2).

(1) Response time

One has an intuitive idea of an interaction (a command is typed, the user waits, output is received from the system) and of response time for this interaction being the period for which the user waits. Although this intuitive idea of response time is generally accepted, a widely-used formal definition of response time has not emerged [PENN84].

The primary goal in measuring response times is to determine the extent to which users have to wait for a system to respond. Interactive performance improves if user waiting is reduced, and worsens if user waiting is increased. Some definitions of response time have been based on attempts to estimate the amount of time during which a user is delayed, waiting for response. One such definition is given by Abrams [ABRA77], who defined response time as "the elapsed time from the last user keystroke until the first meaningful character is displayed". Two problems exist with these *user-oriented* definitions of response time:

(i) In general, it is impossible to determine the instant at which a user ceases to be delayed. For example, in Abrams' definition the "first meaningful character" should be the first character that is meaningful to the user. A performance monitor has no way of knowing which characters are meaningful to a user and which are not. An approximation would be to assume that the first character displayed is the first meaningful character, but the first character displayed may be part of a message such as "Input received" or "Processing started" [GUER83].

(ii) In most graphical user interfaces, one or more windows capable of receiving user input can be displayed at the same time. By directing input to different windows, a user can initiate several interactions that run concurrently. Identifying which interaction the user is delayed by, if any, from a set of concurrent interactions is may be difficult.

To avoid these problems, response time for an interaction can be defined in a *system-oriented way*, as the delay between the instant at which a user input occurs and the instant at which the system processing that resulted from that input finishes. To measure response time defined in this way it is not necessary to determine the period(s) for which a user is delayed by an interaction. In a few situations, it may be preferable to measure response time in a user-oriented way. For example, users who start background programs are concerned primarily with the delay until the system can accept their next command (response time measured in a user-oriented way) rather than the delay until the background program finishes (response time measured in a system-oriented way). In most situations, however, the system-oriented response time of an interaction is of most relevance, and in this thesis the term "response time" is used to mean system-oriented response time.

(2) Response ratio

On their own, absolute values of response time (or mean response times) are meaningless. To determine whether the response time for an interaction represents good performance, it is necessary to have additional information on the complexity of the interaction. For example, a response time of 10 seconds may represent very good performance for an interaction that involved the compilation of a 10,000 line program, and very bad performance for an interaction that involved putting a character into a terminal buffer.

A responsiveness index whose use avoids this problem is response ratio. *Response ratio* for an interaction is the ratio of its actual response time to the minimum possible response time of the same interaction, on the same system [PENN88]. Intuitively, a response ratio of N for an interaction indicates that its response time was N times longer than it would have been had the interaction been executed with no other workload on the system. A response ratio of 2 for an interaction suggests that performance is good and a

response ratio of 100 suggests that performance is poor, regardless of the complexity of the interaction.

A problem for use of response ratio is that the minimum response time for an interaction is usually difficult to estimate accurately unless an interaction can be executed on an otherwise idle system. Doing this is impractical for interactions that occur as part of the normal system workload.

3.4.3 Diagnostic Studies

Diagnostic studies that seek reasons for poor interactive performance are directed at finding "where the response time went". Response time is to be subdivided into various components so that the most significant components can be identified and therapies applied to them. Several techniques for subdividing response times are described below.

(1) Profiling

Profiling is a technique for determining the use of various sections of code during system execution. Therapies can then be applied to heavily used sections of code. Two main types of information are recorded in profiles: counts of events, and times. Counts are of the number of occasions on which statements of interest are executed, with procedure calls commonly counted. A number of ways are available to describe where time is spent during execution. A primitive way of doing this is by dividing the code sections into a number of equal-size regions, and recording the amount of time spent executing in each region [FERR83]. More sophisticated profilers report the processor time spent in each module, and some can even report the total time spent executing all modules called, directly or indirectly, from a module [GRAH82]. Other information that a profiler might produce includes the depth of procedure calls, and procedure call frequencies [KUND86].

(2) Decomposition of response time with respect to resource usage

The resources in question may be: hardware resources, such as processor time, memory, network bandwidth, and device accesses; or software resources such as database records, and critical sections of code [PENN88]. Two main forms of resource decomposition are used. In one form, the time that a program or interaction spends actually *using* each (usually hardware) resource is reported. This information is routinely recorded for accounting purposes in most computer systems. The `time` command of the Berkeley Unix C-shell command interpreter [BERK85] provides a resource usage summary for a program executed from the C-shell. Processor time used, elapsed (response) time, average memory usage, and disc access counts are reported.

The other form of resource decomposition is one in which *all* of the response time of a program or interaction is accounted for. This means that time spent queueing for resources must be recorded, as well as the time spent using resources. Penny and Ashton, [PENN84], [PENN86], [PENN88], and Tetzlaff [TETZ79], [TETZ82] have used this approach. The author developed a sampling monitor for Prime 50 series machines running PRIMOS [ASHT84]. This monitor, ASHMON, measures the time that a process spent in each of the following states: executing on the processor, on one of 4 processor queues, performing a disc access, queued to perform a disc access, performing a magnetic tape access, waiting on a file lock, and waiting on the network. ASHMON determined response time decompositions for the set of interactions performed by one or more processes during the measurement period. Tetzlaff developed a similar monitor for the VM/370 operating system [TETZ79], [TETZ82]. His main interest was to "break the response times into its components, arising from CPU, I/O, and paging", with both queueing and running times being of interest.

(3) Decomposition of response time with respect to function.

Response time can also be decomposed into the times spent performing different functions during an interaction. This approach is applicable only where the interactions all perform similar functions. Mukkamala *et al* [MUKK88] define the following components into which response times of database transactions can be decomposed: data retrieval and computations, network response time for data movement, and overhead due to concurrency control.

3.4.4 Earlier Work

In earlier work, the author developed a performance monitor designed to provide information on interactive performance for a centralised uniprocessor system [ASHT84]. Average response ratio was selected as the performance index to describe interactive performance, and it was used in a number of comparisons. Decomposition of response time with respect to resource usage was used to diagnose the causes of poor interactive performance, as described above in part (2) of Subsection 3.4.3.

3.5 Distributed Systems

Methods for measuring interactive performance for distributed systems are now briefly surveyed, with discussion under the same headings as used in Section 3.4. The nature of user interaction with a distributed system is described in Subsection 3.5.1. The major types of performance information used in comparative performance studies of distributed systems are described in Subsection 3.5.2, and the major types of performance information

used in diagnostic performance studies of distributed systems are discussed in Subsection 3.5.3. More detailed information on some of the tools and techniques referred to is presented in later chapters.

3.5.1 A Description of User Interaction

Users interact with distributed systems primarily through character-oriented and graphical user interfaces, that is in the same ways that they interact with centralised systems. Often, a user of a distributed system sits at a *workstation* that provides a graphical user interface. Resources such as disc space, printing and substantial processing resources are provided by other nodes in the distributed system.

In user interactions with a distributed system, the "system" is much more complex. In a user interaction with a centralised system, all processing is performed on a single node, often by a single thread. In a user interaction with a distributed system, however, processing often involves many different threads, executing on different nodes. The threads that perform a distributed interaction must communicate with each other, and they may run in parallel. The much greater number of components involved in performing a user interaction, and the possibly complex relationships between components, means that evaluating performance for distributed interactions is much more complicated than evaluating performance for interactions with centralised systems [SNOD88]. Tools and techniques for measuring the interactive performance of distributed systems must be able to cope with this complexity.

3.5.2 Comparative Studies

The performance indices response time and response ratio are relevant for distributed interactions, as users interact with distributed systems in the same way that they interact with centralised systems. Estimating response ratio for a distributed interaction is more difficult, as estimating minimum response time for a distributed interaction will be more difficult because of the greater complexity of the distributed system.

A performance index introduced for use in evaluation of distributed systems and multiprocessor systems is speedup (see, for example, [MAND89]). *Speedup* is a measure of how successfully a program makes use of multiple processors, and can be defined as the ratio of the response time of a program executed on N processors ($N \geq 1$) to the response time of that program executed on one processor. A distributed algorithm performs well if speedup with N processors is very close to N, indicating that all N processors perform useful work most of the time.

3.5.3 Diagnostic Studies

Many tools and techniques have been developed for use in diagnostic studies of distributed and parallel systems. Some of these are primarily debuggers that include facilities for performance debugging. The phrase "performance debugging" has been coined (see, for example, [ARAL88]) to describe the activity of improving the performance of a parallel or distributed program to achieve acceptable levels of speedup. Performance debugging is needed because initial versions of parallel and distributed programs often contain bottlenecks that severely limit speedup. Performance debugging is performed mainly for programs for multiprocessors, and Aral and Gertner have commented that "performance debugging on a multiprocessor is as important as functional debugging on a uniprocessor" [ARAL88].

Techniques developed for use in diagnostic studies of distributed systems must allow for the fact that distributed programs can be performed by many communicating threads spread across several nodes. A common way of managing the very large amount of performance data that can be recorded for such a distributed program is to provide performance information in a hierarchical fashion. Information may be available that provides summaries for:

(1) the whole program,

(2) each node used,

(3) each thread,

(4) activities within each thread, such as the execution of program modules.

The IPS [YANG89] and TMP [HABA90] monitoring systems provide performance information in hierarchical frameworks similar to the one outlined. A user can start by examining program summaries, and can then request detailed information on selected nodes, threads, and thread components.

Some types of performance information useful in diagnostic studies are extensions of those types described in Subsection 3.4.3. For example, Aral and Gertner describe a profiler for distributed programs that records profiles for all threads in a distributed program [ARAL88]. Brief surveys of some other types of performance information are given below.

(1) Indices

Performance indices can be useful for some diagnostic studies, particularly when they are available within a hierarchical framework, as in IPS and TMP . Most often, values of indices are plotted against time. Some indices are those that were used for analysis of

centralised systems, such as resource usage times, queueing times and lengths, and file access counts [HABA90], [KERO87], [YANG89]. Other indices have been invented especially for use in analysis of distributed systems, including parallelism (the average number of processors used by a distributed program), speedup, and indices that relate to communication, such as message volume, message size, and message delay [HABA90], [YANG89].

Ball *et al* define the *concurrency graph* for a program to be a graph of parallelism against time [BALL89]. The graph highlights periods of low parallelism that show bottlenecks to be removed if performance is to be improved.

(2) Analysis of communication

Information about time spent in communication is important as:

(i) too many messages or too much message data can cause bottlenecks, and

(ii) incorrectly implemented communication protocols are a common source of errors in distributed programs.

Performance information on the communication that occurs in a distributed program can be presented in many ways, for example:

(i) As a trace. Information on individual messages, shared variable accesses, and synchronisation operations is given in a detailed report. For messages the message source, destination, length, delay, and contents may be given. See for example [JOYC87].

(ii) As summaries. Summaries usually include statistics on message counts and sizes, and might be available at different levels of a program hierarchy with, for example, one summary for the whole program, one for each pair of communicating nodes, and one for each pair of communicating threads. Examples can be found in [JOYC87], [MILL88a].

(iii) As animations. A trace that records details of messages exchanged can be used to produce an animation of the messages passed. A common type of animation is where the communication end-points (nodes or threads) are represented as icons. Each message is represented by some symbol, often an arrow, travelling from the sender's icon to the receiver's icon. For a human to be able to follow an animation, simulated time in an animation must be much slower than real-time. Examples can be found in [HOUG89], [JOYC87], [WITT89].

(iv) As graphical displays. Miller [MILL88a] describes a technique called "hot spots" where colour is used to highlight communication volumes within a distributed program. In this technique, the nodes on which a distributed program is executed are shown as vertices in a digraph, with edges connecting nodes that exchange messages. During program execution, the colour of each edge varies according to the amount of traffic between the

nodes connected by the edge, with edges going from violet to red with increasing traffic. Each edge can be labelled with the value of a performance index that shows the amount of traffic on the communication link represented by the edge. Indices available include messages/second, bytes/second, (cumulative) message counts, (cumulative) byte counts, and average message sizes.

(3) Graph representations

The computation and communication that comprises a distributed computation can be represented in an acyclic digraph. Events relating to computation or communication are represented by vertices, and computation and communication activities are represented by edges. Each edge can be weighted with the duration of the activity that it represents. Every edge represents a forward progression of time, so graphs are acyclic.

An important example of an acyclic digraph representation is the program activity graph (PAG) [MILL90a], [YANG89]. A program activity graph represents one execution of a program on a distributed system. The *critical path* through a program activity graph shows the set of activities such that the length of at least one of the activities must be reduced if response time for the program is to be reduced. Ball *et al* note that "highlighting the critical path is extremely useful to programmers seeking to decrease the execution time of a program" [BALL89].

The IPS monitor has been developed to record program activity graphs, and to perform critical path analysis. The monitor can present components of the critical path of a distributed program, in terms of processor time and message delay times, within a measurement hierarchy of four levels [MILL90a]. At the program level, total processor time and total inter-node message times are reported. At the machine level total processor time for each node and total inter-node message time for each pair of nodes are reported. At the process level, total processor time for each process and total interprocess message time for each pair of processes are reported. At the procedure level, processor time for each procedure executed by a process is reported.

Another example of an acyclic digraph representation is the graphic representation of an IPEL (integrated program-level execution logs) [TSAI90]. Tsai *et al* propose that graphic representations of IPELs be displayed to users to assist in debugging of real-time programs. The program history graph [MILL88a] is another example. All of these representation techniques are discussed in more detail in Section 4.3.

Acyclic digraph representations can record a great deal of information on the execution of a distributed program. Most of the types of performance information described in (1) and (2) can be derived from them.

(4) Summary

The author's research has shown that nearly all tools developed for monitoring of distributed systems are directed toward the analysis of individual programs. Examples are described in [ARAL88], [BALL89], [HABA90], [HOUG89], [KERO87], [MILL88a], [TSAI90], [WITT89], [YANG89]. The JADE monitoring system [JOYC87] does allow for analysis of performance data recorded for some arbitrary set of processes, although usually the set will contain processes that are all part of the same distributed program. In an approach outlined by Snodgrass, analysis is based on performance data collected for all, or selected, processes in a distributed system [SNOD88].

3.6 Summary

The measurement of interactive performance of distributed systems has been discussed in the context of the field of which it is a part: Computer Performance Evaluation, or CPE. Related work on measuring interactive performance, and on measuring performance of distributed systems, has been described.

Our approach to performance measurement of distributed systems will be described in the following chapters. The approach is based on recording an interaction network for each interaction. Definitions of interactions and interaction networks will be given in Chapter 4. The interaction network is an acyclic digraph representation of computation in a distributed system. One important type of performance information available from an interaction network is a summary of the components of response time for the corresponding interaction.

The interaction network approach is unusual in that it is based on the analysis of performance for individual interactions. No other work described in the literature appears to takes this approach.

Chapter 4

The Interaction Network

The user of an interactive computer system conducts a dialogue with the system. The dialogue consists of one or more interactions, where each *interaction* consists of some input action by the user, and some processing and output by the computer system. Because the interaction is the unit of dialogue in an interactive system, we must have an understanding of the nature of interactions if we are to measure the performance of interactive systems. A simple, general model of user interaction is presented in Section 4.1. In Section 4.2, the *interaction network* is introduced as a way of describing the processing performed by a distributed system in response to a user input, and an example is given of an interaction network. The interaction network definition is based on the communicating threads model given in Chapter 2. In Section 4.3, other ways of describing the processing component of an interaction are discussed and compared with the interaction network.

The term "interaction" can have different meanings for different groups within computing. To those interested in human factors and user interfaces, "interaction" is a much broader area, encompassing all issues relating to how humans interact with computers. Major concerns of the latter group include the representation, design, implementation, execution, evaluation, and maintenance of user interfaces [HART89]. In this thesis, though, the term interaction (or user interaction) will be used where the performance-oriented usage of the term is intended, and the term human-computer interaction will be used where the broader human factors-oriented usage is intended.

4.1 A Model of User Interaction

In most models of user interaction that are used in performance evaluation, cycles of user input followed by system output are sequential. That is, after each input a user waits for the resulting system output to finish before providing the next input. In Subsection 4.1.1, these models are summarised and some of their limitations are discussed. A very general model of user interaction that does not have these limitations is then introduced in Subsection 4.1.2.

4.1.1 Other Models of User Interaction

Most definitions of a user interaction are based on the sequence: the user inputs a request to the system, which processes the request, and then passes output from the request back to the user. The user then thinks for a period, before inputting the next request. This view of an interaction is prevalent in both measurement ([FERR83] and [SHNE84], for example), and modelling ([HEID84] and [LEUN88], for example). Closely bound to the concept of an interaction is the idea of *response time*, an important performance index for interactive systems that was described in Subsection 3.4.1.

Many definitions of an "interaction" are based on interfaces to interactive systems provided by character-based terminals. Shneiderman's definition [SHNE84], illustrated in Figure 4-1, is typical.

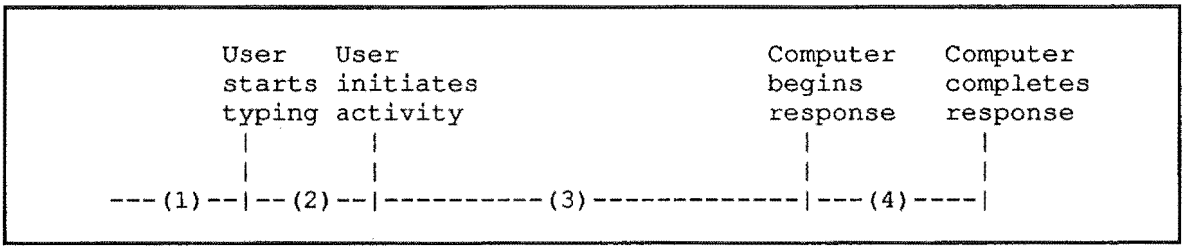


Figure 4-1: Shneiderman's description of a user interaction

The periods of time numbered (1) to (4) in Figure 4-1 are:

(1) Think time

(2) Input time

(3) Response time

(4) Output time

A user may initiate an activity by pressing the return key, pressing a "start processing" key, or by doing something similar. Ferrari [FERR78] describes an interaction model very similar to that of Shneiderman, and defines *interaction time* as the time for a complete *interaction cycle*, that is the sum of the times for (1), (2), (3), and (4) in the example above.

This transaction-oriented view of user interaction, also known as "sequential dialogue" [HART89], is not general enough to describe user interactions where modern graphical user interfaces are used. These interfaces allow output of graphics as well as text, allow mouse, touch-screen and voice input as well as keyboard input, and allow the user to be in control of many interactions simultaneously [HART89]. Clearly, a more general model of user interaction is required.

4.1.2 A New Model of User Interaction

The user interaction model presented here is both simple and general. The model is based on the idea that an interaction is the pair:

<user action, system reaction>

A user action is the input of a primitive information carrying unit, or *lexeme* [NIEL86], by the user, and the reaction is **all** of the processing (which may include output) resulting from the input of the lexeme.

A lexeme may result from a key-stroke for input from a keyboard, a mouse movement or mouse button-up or -down event for input from a mouse, and so on for other types of input, including touch-screen input, voice input, video input, automatic teller machine input, and bar code reader input. If it is possible to input a lexeme while processing for a previous lexeme is still in progress, then either the input lexeme may be buffered until the processing of the previous lexeme has finished, or it may be processed concurrently with the previous lexeme.

Defining an interaction in terms of the most primitive user actions gives two advantages over the transaction-oriented definition of an interaction. First, it emphasises the fact that even for the simplest interactions there may be questions of performance. In noting the costs where response times are greater than user expectations, Foley and van Dam [FOLE82] discuss "reflex actions", such as typing a key or moving the cursor, for which response times should be less than 100 milliseconds. They suggest also that the time taken to respond to "simple" interactions should be no more than about 2 seconds.

The second advantage is that more complex <action, reaction> combinations can be defined in terms of primitive interactions, thus allowing a much greater range of user interface styles to be represented. Transaction-oriented interfaces can be described as follows. Each character input on a command line is a lexeme, with an associated system reaction. For a command requiring the input of n characters, the first n-1 system reactions are trivial, usually the insertion of a character into an input buffer. The nth character, a 'newline' for instance, initiates the command. The system reaction associated with the nth lexeme includes all of the processing performed during periods (3) and (4) in Figure 4-1.

The model can also represent *asynchronous dialogue* [HART89], where the user can be involved in several tasks at any one time, and interactions may be taking place in parallel. Each interaction is still initiated by the input of a lexeme, but now system reactions can overlap. So, for asynchronous dialogue, the set of interactions that a user initiates while logged into a system, the user's *session*, can be characterised as a set of possibly overlapping interactions, each initiated by the input of a lexeme. For sequential dialogues,

such as those provided by command-line interfaces, a user's session can be characterised as a set of non-overlapping interactions, each initiated by the input of a lexeme.

Similarities exist between the <user action, system reaction> model of a user interaction, and other models described in the literature. In van Renesse's functional processing model [RENE89], each user input causes the termination of the current process and the creation of a new one. Benbasat and Wand [BENB84] describe "a model for human-computer interaction" based on a command-line interaction style. Their model is "based on a dialogue being viewed as a sequence of basic interaction events", where "an interaction event is defined as an occurrence in the dialogue where the system awaits input from the user". The processing of an interaction event consists of prompt, input, action, and flow control (the selection of the next interaction event). They also state that "all processing that occurs between two interaction events is considered as the action of the first event". This model is similar to our own, where the input is a user action, and the processing that occurs between two interaction events is the system reaction to the first event. The main difference is that our model is more general, in that it can allow for all types of user interface, including graphical user interfaces, whereas their model is restricted to the command-line style of user interaction.

4.2 The Interaction Network

The *interaction network* is an acyclic digraph representation of the way in which a distributed system performs a system reaction to a user action, based on the <user action, system reaction> model of user interaction introduced in Section 4.1 and the communicating threads model described in Chapter 2. An interaction network can be used to represent system reactions performed by single or multiple threads, and to record use by those threads of hardware, software, and data resources.

In this section, the interaction network concept is described in detail. The concepts of tasks and sub-tasks, needed to define the interaction network, are discussed in Subsection 4.2.1. The meanings assigned to edges and vertices in an interaction network are described in Subsection 4.2.2, and in Subsection 4.2.3 the different types of event represented in an interaction network are introduced. In Subsection 4.2.4, we describe how an interaction network can be used to represent processing for the communicating threads model. An example is given in Subsection 4.2.5, together with a layout method for display of interaction network graphs.

4.2.1 Tasks and Sub-tasks

In Chapter 2, computation in a distributed system was defined in terms of communicating threads and, in Section 4.1, a user interaction has been defined as a <user

action, system reaction> pair. Each user action causes the system to perform a *task*, which is the system reaction to that user action. A task is created when the system recognises a user action; for example the system may detect a key-stroke. Recognition of the user action results in a thread receiving notification of the occurrence of the action. Processing for the task starts in this initial thread, and may spread via communication to many other threads. The task finishes when the last thread performing processing for the task completes that processing. Some threads which perform processing for a task may already exist, while others may be created during the course of the task. Some threads that perform processing for a task may terminate during the task, while others (such as threads in a server) remain to perform processing for other tasks.

Each task is performed by communicating threads. To provide a common framework for representing thread execution and thread communication the idea of a sub-task is now introduced. Each task is carried out as one or more sub-tasks, where a *sub-task* is such that the steps in a sub-task are performed in a wholly sequential manner. At any instant, a sub-task is associated either with a thread performing work for a task, or with a message associated with a task. The number of sub-tasks within a task varies dynamically, but is always at least one throughout a task's existence.

When a user action is recognised a sub-task is created. This sub-task is then associated with the thread that does the initial processing. As a task progresses, sub-tasks are created as needed, and each terminates when it has completed its part of the task. Because each sub-task represents a sequential part of a task, a new sub-task must be created whenever a new thread is created, or a message is sent. A sub-task terminates when the thread or message with which it is associated has completed its work for the task, or when it joins with another sub-task. A task is finished, and the system reaction completed, when the last of its sub-tasks terminates.

4.2.2 Representing the Execution of a Task

The processing performed by the sub-tasks of a task is represented by an interaction network. Each vertex v_i of an interaction network represents an *event* that occurred at time(v_i) in the life of a sub-task. Each edge joining two vertices in an interaction network represents an activity performed by a sub-task.

Every edge is directed, and represents a forward progression in time. That is, for edge (v_i, v_j) that goes from vertex v_i to vertex v_j, time(v_i) < time(v_j). Each sub-task, then, is represented by a *line*, defined as a connected sequence of vertices representing the events within the sub-task. An interaction network consists of one or more interconnected sub-task lines.

An interaction network should:

(1) Record the occurrence of events that mark the beginning and the end of each sub-task. We will call these events *structural events*.

(2) Record the occurrence of other, *performance-relevant*, events which occur during a sub-task. A few examples of events likely to be of interest for performance analysis are: allocation of a processor to a thread, the beginning and the end of a disc request, the appending of a message to a message queue, procedure calls and returns, the opening and closing of files, and accesses to data elements in a database. The set of performance-relevant events that can be recorded will vary between different monitors that record interaction networks. Also, the types of performance-relevant event recorded in a particular measurement session can be selected by a performance analyst.

4.2.3 Representing Events in an Interaction Network

Five patterns of edge connection to vertices, as illustrated in Figure 4-2, are capable of representing nearly all events to be recorded in an interaction network. Source and fork vertices represent the creation of a new sub-task (the number of sub-tasks is increased by one), sink and join vertices represent the termination of a sub-task (the number of sub-tasks is decreased by one), and simple vertices represent the occurrence of other, performance-relevant, events (the number of sub-tasks is unchanged). The few sorts of event that need to be represented by other vertex types are described in Subsection 4.2.4.

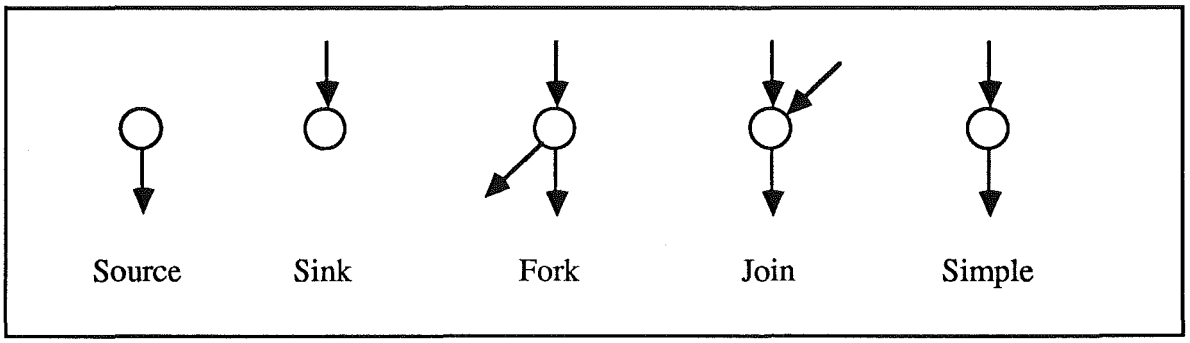


Figure 4-2: Basic vertex types found in interaction networks

In the discussion below, reference must be made to the various incoming and outgoing edges of each vertex type. Where there is more than one incoming or outgoing edge, there must be some way of naming each edge. A fork vertex has two outgoing edges, one of which represents an activity in the newly created sub-task, and one of which represents an activity in the creating sub-task. These edges, then, represent the *created* sub-task and the *creating* sub-task. A join vertex has two incoming edges, one of which represents an

activity in the sub-task that terminates at the join vertex, and one of which represents an activity in the sub-task that continues on from the join vertex. These edges, then, represent the *terminating* sub-task and the *continuing* sub-task.

Saying that "the edge represents an activity in sub-task X" is cumbersome, and in the following we will say simply that "an edge represents sub-task X".

(1) Sub-task creation events

A sub-task can be created in one of two ways:

(i) When a user action is recognised. This event is represented by a source vertex, and there is only one source vertex in each interaction network. The outgoing edge represents this new sub-task.

(ii) When a sub-task creates a new sub-task. This event is represented by a fork vertex, with the incoming edge and one of the outgoing edges representing the creating sub-task, and the second outgoing edge representing the created sub-task. Such events most commonly occur when an existing thread creates a new thread, or when a thread sends a message. The new sub-task is associated respectively with the new thread or the new message. A "send message event" may be a message send, a shared variable write, or a synchronization operation.

(2) Sub-task termination events

Analogously, a sub-task can end in one of two ways:

(i) When a thread or message has completed the work it has to perform for the task. This event is represented by a sink vertex, with the incoming edge representing the sub-task which ends at the sink vertex.

(ii) When two sub-tasks join. This event is represented by a join vertex, with one of the incoming edges and the outgoing edge representing the continuing sub-task, and the second incoming edge representing the terminating sub-task. Such events occur, for example, when two threads join or when a thread receives a message. A "receive message event" may be a message receive, a shared variable read, or a synchronization operation.

(3) Other events

All other types of event do not affect the number of sub-tasks in existence, and are represented by simple vertices.

4.2.4 Relationship to the Distributed Systems Model

The communicating threads model defined distributed systems in terms of nodes, threads, and communication mechanisms (message passing and shared memory). This subsection describes how these constructs are represented in interaction networks.

(1) Nodes

Each thread executes on a node. Node usage can therefore be deduced from information recorded about the threads used by a task.

(2) Threads

During its existence, a thread is associated with one or more sub-tasks at different times. Often, a thread is associated with a single sub-task for the entire execution of the thread. Other threads are associated with many sub-tasks during their execution. For example, a thread that is part of a file server will be associated with a different sub-task for the period of each of the requests that it performs. Instances of both types of thread appear in the example of an interaction network given later.

An interaction network shows the association between a sub-task and a thread, including the beginning of the association, the period of association and the end of the association.

(i) Beginning. A sub-task can become associated with a thread in one of two ways. First, when a thread is created it has a sub-task associated with it. This event is represented by a fork vertex, with a new sub-task being created and associated with the new thread. Second, when a thread receives a message (which has an associated sub-task), the sub-task associated with the message switches to being associated with the receiving thread. This is represented by a simple vertex or a join vertex.

Information on the identity of the thread, its execution context (the cluster, process, or object it is executing in for example), and the node the thread is executing on, are among the things that can be recorded when the "beginning" event occurs. If a thread can migrate to another node part-way through its execution [SMIT88], then a thread might change nodes during the period of association. Each migration can be represented as a series of one or more connected vertices, so if a thread migrates, the node to which it migrates can be recorded as a parameter of one of these events.

(ii) Period. During the period that a sub-task is associated with a thread, any events of interest that occur during the thread's execution are recorded as events that occurred within the sub-task. These events are represented by a series of fork (creating role), join (continuing role), and simple vertices as appropriate.

(iii) End. The end of an association between a thread and a sub-task is shown either as a sink vertex or as a join vertex (terminating role).

The creation of a thread is represented by a fork vertex, and the termination of a thread is represented by a sink vertex.

In some situations, a sub-task is effectively put into a queue by a thread for a period. For instance a file server thread may act in the following way. A request message is received from a client asking that a file read be performed. The file server thread schedules a disc read on behalf of the sub-task, and then performs processing on behalf of other requests until the disc read completes. During the period of the disc read, the sub-task associated with the request is queued, waiting for the file server thread to resume work on its behalf. This is a situation where the sub-task is associated with a thread for two separate periods, with a queuing period in between. Originally it was thought that a "queuing" sub-task state was needed in addition to the "message" and "thread" states. However, the queuing state can be represented as the thread sending a message to itself.

(3) Message passing: message boundaries preserved

Message boundaries are preserved in some message passing protocols, and are not preserved in others. Where message boundaries are preserved the data sent in a message is received in a single unit by the receiving thread. Where message boundaries are not preserved then it is possible for a thread to receive in a single operation data that was sent in several different send operations, and for data from a single send operation to be split up and received in several different receive operations. We now consider the representation in an interaction network of message passing protocols that preserve message boundaries. In (4) we consider representing message passing where message boundaries are not preserved.

Communication involves at least two events: the sending thread performing some sort of send operation, and the receiving thread performing some sort of receive operation. The message send is represented by a fork vertex, as a new sub-task is required to accompany the message. If the receiving thread is associated with a sub-task that is part of the same task as the sub-task associated with the message, then the receive event is represented by a join vertex. Otherwise, the receive event is represented by a simple vertex. In both cases the sub-task associated with the message becomes associated with the thread and the sub-task that was associated with the thread terminates, as it is assumed that on reception of a message a thread finishes what it was doing (the thread sub-task), and starts operating on the data included in the message (the message sub-task).

Other message-related events may occur between the send and the receive events. It is often the case that a message is received at a node and put into a message queue before

being received by a thread. If this occurs, then the event of the message being enqueued is represented by a simple vertex in the sub-task of the message. If the message passing is indirect, then the events of the message arriving at the mailbox and leaving the mailbox can be represented by simple vertices. If the message passes through several nodes, as it would do in a store-and-forward network for instance, and processing in the intermediate nodes is of interest, then there would be several vertices in an interaction network representing events that show progress of a message through the communication network.

In Chapter 2, both blocking and non-blocking variants of send and receive were discussed. As well, it is possible either for a message to arrive before the receiver tries to receive it, or for the receiver to try to receive a message before the message has arrived. Thus there are 8 combinations of the three variables: send type (blocking or non-blocking), receive type (blocking or non-blocking) and first arrival (message or receiver) to consider. Non-blocking receives are not considered further because:

(i) if a non-blocking receive is performed when no messages are available for reception, then the thread that attempted the receive continues on without having received a message, and the non-blocking receive is represented by a simple vertex, and

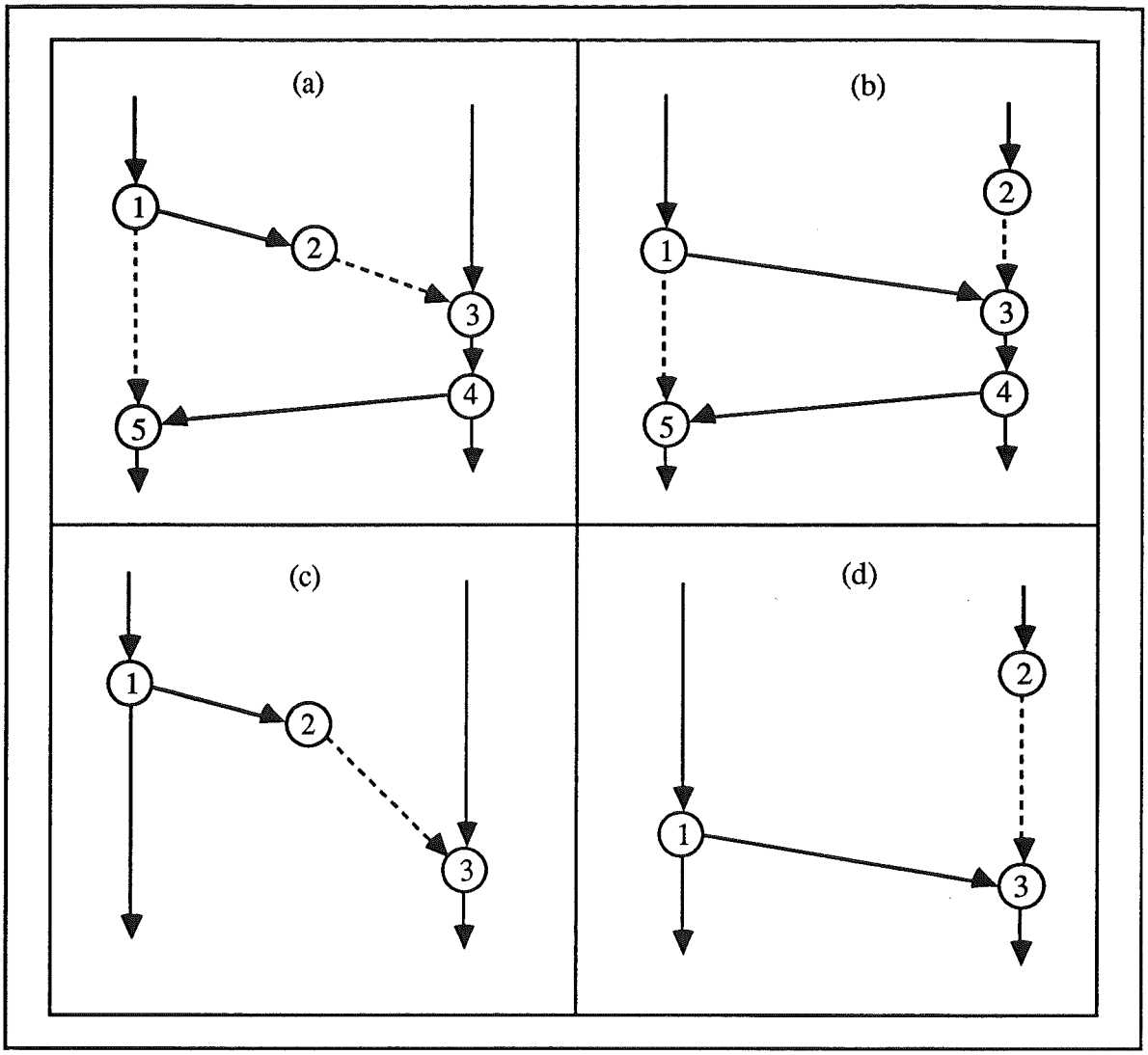
(ii) if a non-blocking receive is performed when a message is available, then the interaction network fragment to represent this is identical to the one for a blocking receive.

Figure 4-3 shows the remaining four cases, with the message send represented in each case by vertex v_1. In the figure, (a) and (b) show synchronous, and (c) and (d) asynchronous, message passing.

(a) Blocking send, message arrives first. The message is sent at v_1 and the sending thread is blocked. At vertex v_2 the message is received by the destination node, but the receiving thread is not waiting to receive it. The time that the message spends waiting is represented by edge (v_2,v_3), that is the message is delayed for some period of time equal to $time(v_3) - time(v_2)$. The receiving thread performs a receive at vertex v_3, an unblocking message is sent to the sending thread (edge (v_4,v_5)), and the sending thread is unblocked at vertex v_5.

(b) Blocking send, receiver arrives first. All vertices except vertex v_2 represent the same events as in (a) above. The receiving thread performs a blocking receive at vertex v_2, and remains blocked (edge (v_2,v_3)) until the message arrives and is received immediately at vertex v_3.

(c) Non-blocking send, message arrives first. The message is sent at vertex v_1 and the sending thread continues. The message arrives at its destination node at vertex v_2, but remains queued (edge (v_2,v_3)) until the receiving thread performs a receive at vertex v_3.



(a) Blocking send, message arrives first.
(b) Blocking send, receiver arrives first.
(c) Non-blocking send, message arrives first.
(d) Non-blocking send, receiver arrives first.
(Blocking shown by dashed lines.)

Figure 4-3: Combinations of send and receive arrival order

(d) Non-blocking send, receiver arrives first. The message is sent at vertex v_1 and the sending thread continues. The receiving thread performs a blocking receive at vertex v_2, and remains blocked (edge (v_2,v_3)) until the message arrives and is received immediately at vertex v_3.

Higher level message passing protocols, like remote procedure call, Ada rendezvous, and CSP message passing [ANDR83] are implemented using the blocking and non-

blocking message send and receive primitives. As message passing using these primitives can be represented in an interaction network, message passing using these higher level protocols can also be represented.

In some message passing protocols where message boundaries are preserved, messages can be sent to several threads by a single (broadcast or multi-cast) send operation. A broadcast or multi-cast send can be represented by a multi-way fork vertex, which has one outgoing edge for each thread that receives the message.

(4) Message passing: message boundaries not preserved

To represent in an interaction network the case of message passing where message boundaries are not preserved, we must be able to show message joining and splitting. In Figure 4-4, a way is shown by which message joining and splitting can be represented is shown. The representation of a thread receiving or sending messages is the same as described above in (3).

Joining and splitting can occur at the node of the sending thread and/or the node of the receiving thread, with joining and splitting occurring at the node of the receiving thread in the example given. In Figure 4-4, the thread *Sender* is sending data to the thread *Receiver* using a form of message passing that does not preserve message boundaries. *Sender* sends 4 messages (vertices v_1, v_2, v_3, and v_4), each of (say) 20 bytes. Each message arrives at the node on which *Receiver* is executing, and is put into a buffer (vertices v_5, v_6, v_7, and v_8). After a time, *Receiver* tries to read 50 bytes. It receives 50 bytes of data, which consists of all of the first two messages, and the first 10 bytes from the third, with the remainder of the third message left in the buffer. This joining and splitting of messages is represented by vertex v_9, and the reception of the 50 bytes is represented by vertex v_{11}. Later, *Receiver* requests a further 50 bytes. This time 30 bytes are available, 10 from the third message and 20 from the fourth. The joining of these messages is represented by vertex v_{10}, and the reception of the 30 bytes is represented by vertex v_{12}.

To represent joining and splitting of messages, we have introduced a new type of vertex. This vertex type has one or more incoming edges (representing the messages being joined), and one or two outgoing edges (representing the message formed by the join, and the message fragment that remains, if any). A newly created sub-task is associated with the message formed by the join. If a message fragment remains then it is stays associated with its original sub-task. For example, edge (v_9,v_{10}) represents the period of time that the last 10 bytes of the third message spends in the buffer after the first 10 bytes of the third message is received. It is associated with the sub-task created for the third message at vertex v_3.

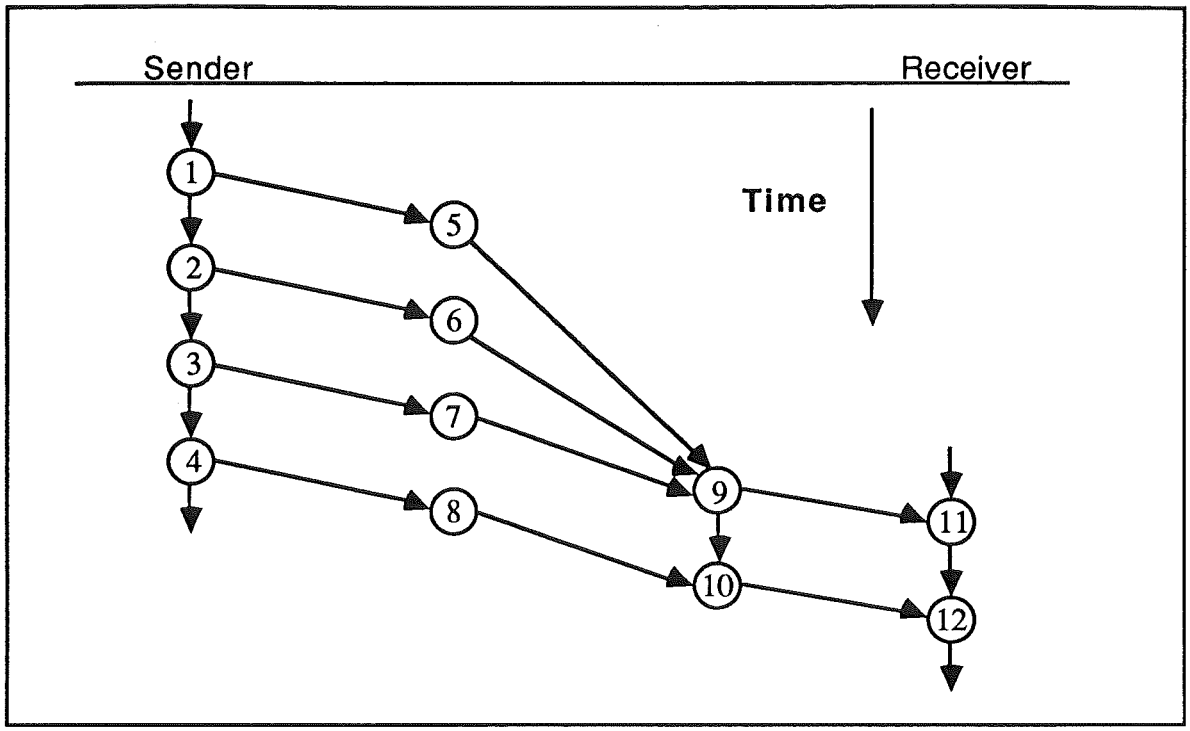


Figure 4-4: Example of message passing where message boundaries are not preserved.

If all of the sub-tasks that join at a message join/split vertex are part of the same task, then it is clear that the created sub-task should also belong to this task. If the joining sub-tasks are, however, part of different tasks, then a choice must be made as to which task the created sub-task is part of. Although in practice this seldom occurs, a way of handling it must be found. We have decided, somewhat arbitrarily, that the created sub-task should be in the same task as the sub-task associated with the last message fragment included in the message received. To see why this choice is reasonable, consider the case where terminal input characters are accumulated in a buffer, and a read from the buffer by a thread is completed only when a whole line of input is available. Each character in the buffer is associated with a different task, as each is the result of a different user input. When a read occurs, all of the sub-tasks of the characters in the line read are joined. It seems reasonable that the sub-task of the input line read be part of the task of the 'newline' that terminates the line, and the 'newline' is the last character in the buffer read.

(5) Shared variables and synchronisation

There are two important differences between the characteristics of message passing and shared memory as communication mechanisms.

(i) In communication by message passing, messages carry data between threads, and also synchronise threads because a message cannot be received until after it has been sent. In communication through shared memory, shared variables provide data transfer, but do not enforce synchronisation, so separate synchronisation primitives are required.

(ii) Messages are transient, with each message being "written" once by the sender, and "read" once by the receiver. Shared variables, on the other hand, can exist for much longer periods, and may be read and written many times. Where a shared variable is written and then read more than once before the next write, the communication style is very similar to broadcast and multi-cast communication in message passing, and can be represented in the same way.

Difference (i) means that the representation in an interaction network of use of shared variables and of synchronisation primitives must be considered separately.

The writes and reads of shared variables can be represented in a similar way to the sends and receives of messages. One difference is that shared variable reads and writes are always non-blocking, as a write changes the variable immediately, and a read returns the current value. A shared variable write is represented by a fork vertex in the same way as a message send. A shared variable read is represented by a join vertex or a simple vertex in the same way as a message receive. Because a shared variable read always reads a value that is already available, shared memory communication is similar to that shown in Figure 4-3c. Vertex v_1 is the write of the shared variable, and vertex v_3 the read of the shared variable. However, vertex v_2 can be removed entirely, as the message is available immediately. The length of time that the value written spends waiting to be read may be small, but is always non-zero.

Several synchronisation primitives have been defined in the literature [ANDR83]. All provide basically the same operations:

(i) wait, if necessary, until a shared resource is available.

(ii) acquire the resource

(iii) use the resource

(iv) release the resource

Operations (ii) and (iv) above are events, and are therefore represented in an interaction network by vertices. The period of use (iii) is represented by a subgraph (edges and vertices) that connects the vertices representing (ii) and (iv). If a thread must wait for the resource, as in (i), then the beginning of the wait is represented by a vertex, which is connected to the vertex representing (ii) by an edge that represents the time spent waiting for the resource.

Synchronisation primitives allow a group of threads to communicate to the extent that the members of the group can coordinate their execution. This communication should be representable in an interaction network. The communication that occurs between a thread that releases a resource and the next thread to acquire the resource can be regarded as a message from the releasing thread to the acquiring thread. Unlike messages, however, it is not unusual for a release (the send equivalent) and the next acquire (the receive equivalent) to occur in different tasks. This is because the activities being synchronised may be otherwise independent of each other, and therefore may be from separate interactions. So, if both the releasing and acquiring sub-tasks are part of the same task, the release and acquire events are shown as message send and receive events; if not then each event is represented by a simple vertex, with the release and acquire events in different interaction networks.

For a join vertex where a "resource available" message is received, the message sub-task terminates and the thread sub-task continues, the reverse situation of that where a data message is received. The reason is that when a thread receives a message containing data, it is assumed that the thread finishes what it was doing, and starts operating on the data included in the message (the message sub-task). When a thread receives a "resource available" message, however, this simply notifies a thread that it can now continue with what it was doing before (the thread sub-task).

"Resource available" messages have a non-blocking send, as a releasing thread need not wait for any sort of "acknowledgement" from the next acquiring thread. Therefore the subgraphs shown in Figures 4-3c and 4-3d can represent the use of synchronisation primitives if the release and acquire are within the same task. In both figures, vertex v_1 is the release, and vertex v_3 is the next acquire. In Figure 4-3c the release occurs before the acquire so the release "message" is blocked until the acquire occurs. In Figure 4-3d the acquire occurs before the release, so the acquiring thread must wait (edge (v_2,v_3)) until the resource is released.

(6) Summary

We have shown that the events and activities performed by communicating threads during a system reaction can be represented in an interaction network. The five vertex types shown in Figure 4-2 are sufficient to represent almost all events that occur during a system reaction. Two other vertex types are required, one to represent message joining and splitting in message passing where message boundaries are not preserved, and one to represent broadcast and multi-cast sends.

Some assumptions have been made about which task the outgoing edge of a join vertex belongs to in cases where the incoming edges belong to different tasks. The following choices have been made:

(i) For a message receive and shared variable read, the task of the outgoing edge is the same as that of the message.

(ii) For a "resource available" message, the outgoing edge is the task of the thread.

(iii) For a message join, the task of the outgoing edge is the task of the last of the messages included as part of the message that results from the join.

Intuitively, these choices seem reasonable, and they have worked well in the prototype monitor, INMON.

4.2.5 The Interaction Network

The interaction network can now be considered as a whole. Each sub-task begins at either the source vertex (the initial sub-task), or a fork (created role) vertex (all other sub-tasks), and progresses through a series of fork (creating role), join (continuing role), and simple vertices until a sink or join (terminating role) vertex is reached. The interaction network shows all of a sub-tasks of a task, the connections between sub-tasks, and simple vertices representing other performance-relevant events of interest.

An interaction network is a (weakly) connected acyclic digraph D = (V, E), where V is the set of vertices and E the set of edges in the digraph. An interaction network is a weakly connected digraph as all vertices can be reached from the source vertex. This is because: all vertices in a sub-task are connected; all sub-tasks created after the first are either created directly by the first sub-task, or are created by one of its descendants; and each created sub-task is connected to the sub-task that created it. An interaction network is acyclic because each edge represents a forward progression of time, and a cycle would require at least one edge that went back in time, or at least two edges which went neither forward nor back (sideways perhaps) in time.

The function time(v_i) gives the time at which the event represented by vertex v_i occurred. Although it is difficult to provide a global clock in a loosely-coupled distributed system, many methods have been developed for keeping the clocks in a distributed system closely synchronised. For now, we assume that event times are consistent with the partial order [LAMP78] of events, and that the clocks in a distributed system are closely synchronised. Chapter 8 discusses clock synchronisation in more detail.

The graph for an interaction network can be drawn in a systematic way, as will be illustrated in Figures 4-6, 4-7, and 4-8. The position of each vertex is computed as follows:

(i) The Y-coordinate. Assume that there are n vertices in the graph, v_0, v_1, ..., v_{n-1}. The initial vertex v_0, the vertex corresponding to the recognition of the user action, is at the top, with time increasing down the page. If a performance analyst wants an interaction network display to show the relative duration of each activity, then the Y-coordinate of some vertex v_i must be calculated as a linear function time(v_i). If relative durations of activities are not important, then the Y-coordinate of each vertex can be determined in some other way. For example, it has been found when recording interaction networks in practice that over some periods events occur in very quick succession. If a linear function of time is used in determining vertex coordinates, then the vertices that represent events in each group are very tightly clustered. To separate tightly clustered vertices, the Y-coordinate of each vertex could be determined by having a fixed minimum spacing between vertices that represent consecutive events, as in the "fixed" layout method described in Subsection 10.5.2.

(ii) The X-Coordinate. The threads involved in the interaction are listed across the top of the page. Each vertex representing a thread-related event is put underneath the name of the thread in which it occurred. Vertices that represent communication events are placed between the columns of the sending and receiving threads.

(1) Example

An example of an interaction network is now given for a simple interaction. Consider a small distributed system which consists of three nodes: an Apple Macintosh, a discless Sun workstation (kaka) and a Sun workstation with local disc (kiwi). All three are connected via ethernet. A person is using the NCSA Telnet terminal emulator on the Macintosh. A single thread is executing the NCSA Telnet program, through which the user has initiated a login session on kaka. The login has created two Unix processes, *telnetd* and *csh*. The *telnetd* process is responsible for relaying inputs and outputs between *NCSA Telnet* on the Macintosh and, on kaka, a *pseudo-terminal* containing input and output buffers. *csh* is a command interpreter which receives input from, and delivers output to, the pseudo-terminal. When the user types a character on the Macintosh keyboard, *NCSA Telnet* sends it across the ethernet to the *telnetd* process which adds the character to the pseudo-terminal input buffer. *csh* subsequently reads the character from the pseudo-terminal.

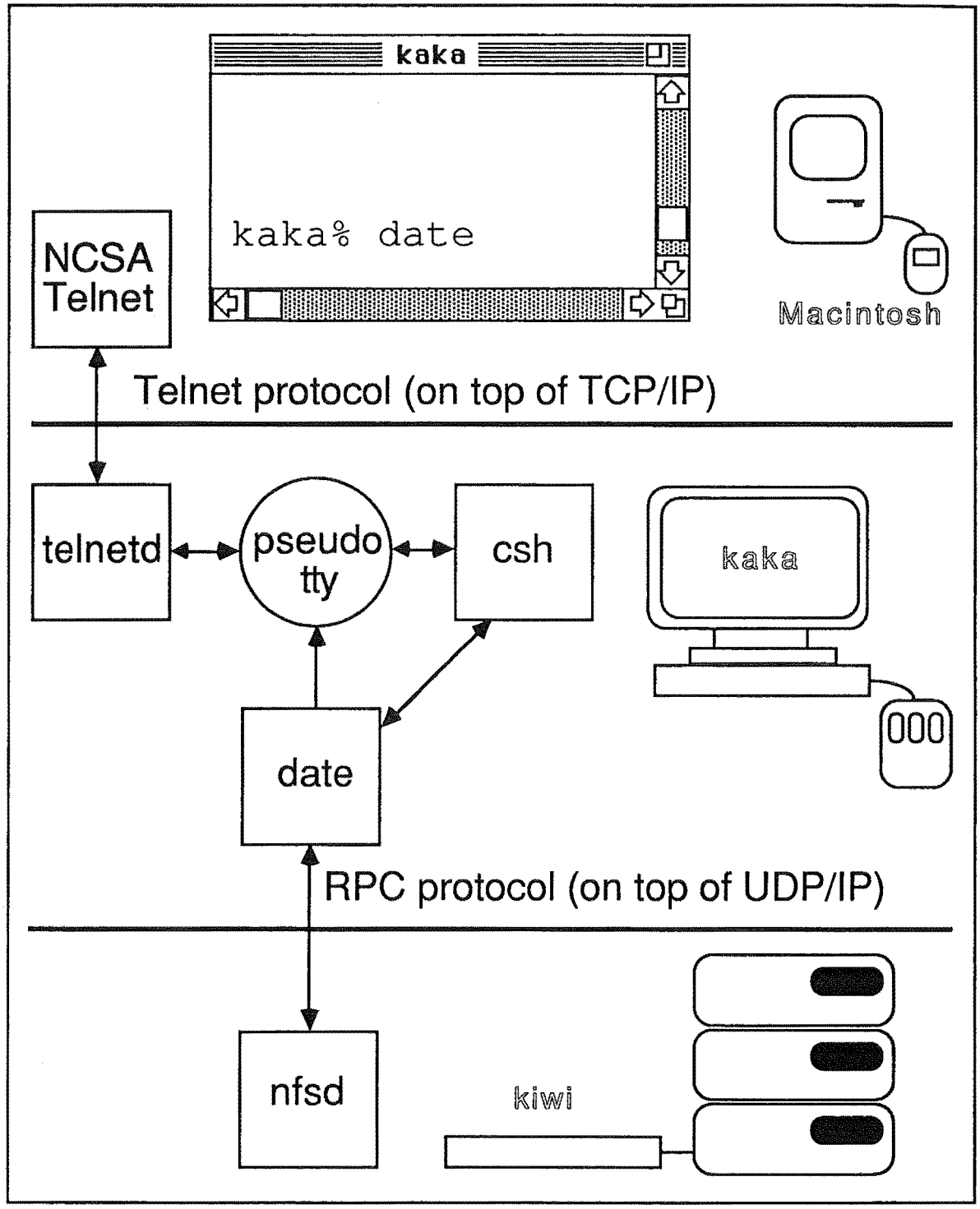


Figure 4-5: System overview for interaction network examples

The user wishes to enter the Unix command "date", which prints out date and time information. To do this, the user inputs 5 lexemes, by typing 'd', 'a', 't', 'e', and 'newline'. When 'newline' is added to the pseudo-terminal input buffer, "date\n" is read by *csh*, which then creates the *date* process. *date* determines the current date and time and

writes it to the pseudo-terminal. An overview of the distributed system and the interaction is shown in Figure 4-5. In the figure, *date* is shown communicating with *nfsd* on the file server kiwi. Such communication would occur if, for example, a page fault occurred during the execution of *date*, requiring *date* to request a page transfer from the file server kiwi, which would be handled on kiwi by an *nfsd* process. In the example that follows, it is assumed that no such file server requests are necessary, but file server accesses are considered later in this subsection.

Figure 4-6 shows the interaction network for the 'newline' lexeme. The interaction networks for input of the 'd', 'a', 't', and 'e' lexemes all have the same form as each other, that is, vertices v_1 to v_9 of Figure 4-6 and the edges that connect them.

In the system reaction for the 'newline' lexeme, use is made of two nodes and four threads: one thread on the Macintosh executing in *NCSA Telnet*, and three threads on kaka: *telnetd*, *csh*, and *date*, each of them a Unix process.

The sub-tasks represented in the interaction network in Figure 4-6 can be easily identified. Fork and join vertices have two associated sub-tasks, all other vertices have one associated sub-task.

Sub-task vertices	Function
v_1, v_2, v_3	The 'newline' is received, and sent to *telnetd*. *NCSA Telnet* then waits for the next event.
v_2, v_4, v_5, v_6, v_7	The 'newline' is sent to *telnetd*, received, and enqueued in the pseudo-terminal input queue. *telnetd* then echoes the 'newline', and waits for the next event.
v_6, v_8, v_9	The echoed 'newline' is sent to *NCSA Telnet*, which prints it in the appropriate window, and then waits for the next event.
$v_5, v_{10}, v_{11}, v_{12}, v_{13}$	The 'newline' is placed in the input queue of the pseudo-terminal. The *csh* process reads the command line (which is "date\n"), creates the *date* process, and waits for *date* to complete.
$v_{11}, v_{16}, v_{17}, v_{18}$	*date* determines the current date and time, prints them and then terminates, causing a message to be sent to *csh* informing it that *date* has terminated.
$v_{17}, v_{13}, v_{14}, v_{15}$	A process termination message is sent from *date* to *csh*. *csh* resumes execution, prints a prompt, then waits for the next input.

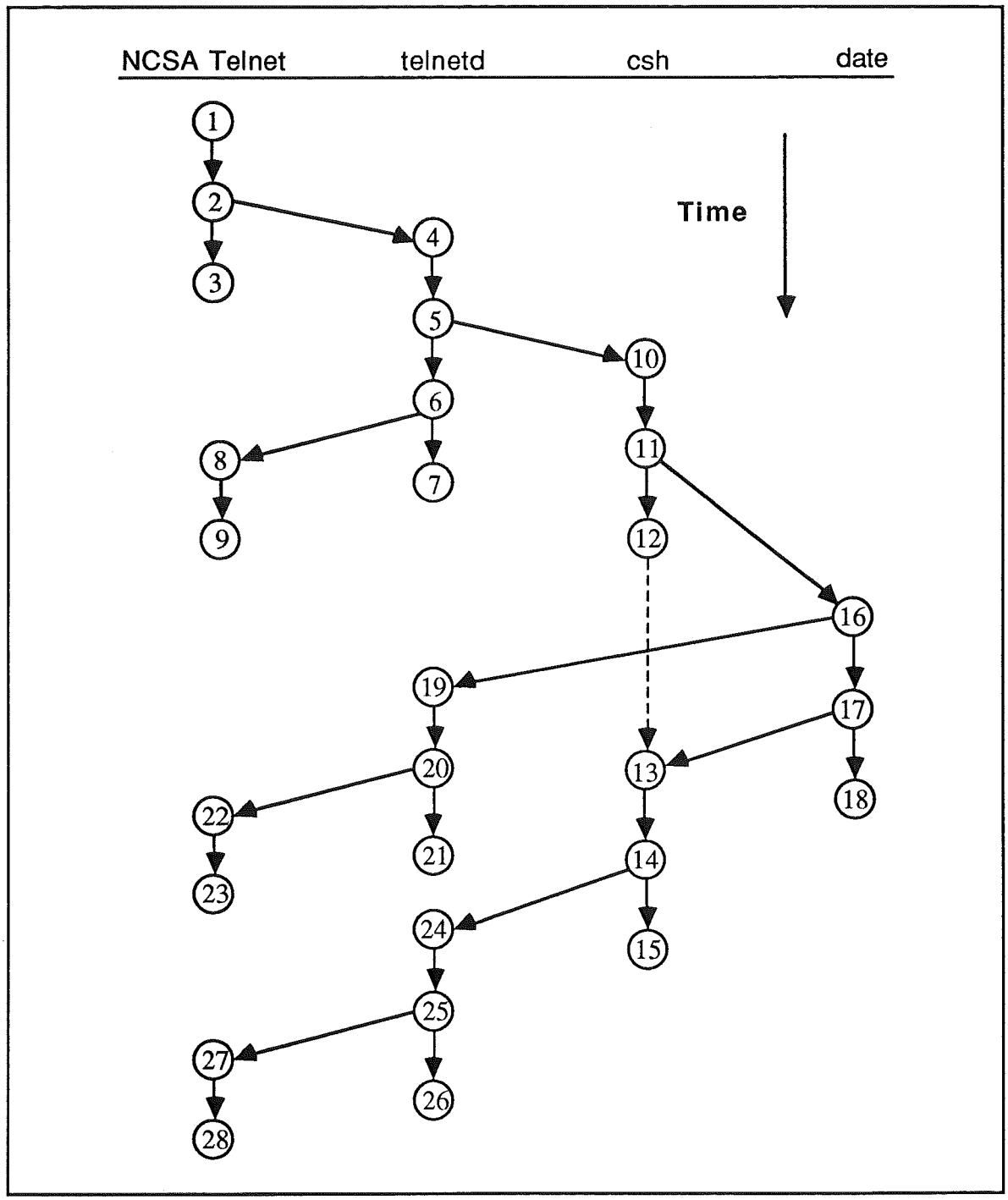


Figure 4-6: Interaction network for the 'newline' lexeme

v_{16},v_{19},v_{20},v_{21}	The date and time output is received by *telnetd* (via the pseudo-terminal), and then sent to *NCSA Telnet*. *telnetd* then waits for the next event.
v_{20},v_{22},v_{23}	The date and time is received by *NCSA Telnet*, which prints the output in the appropriate window, and then waits for the next event.
v_{14},v_{24},v_{25},v_{26} & v_{25},v_{27},v_{28}	These two sub-tasks cause the *csh* prompt to be printed, and work in the same way as the two previous sub-tasks.

Each edge represents either computation or communication. Vertical edges (and edge (v_{11},v_{16})) represent computation, while non-vertical edges (except for edge (v_{11},v_{16})) represent communication. (To achieve consistency, we could use a vertex v_{11a} to imply a virtual message from *csh* to *date*, but that would not accurately represent what actually happens in the Sun operating system.)

(2) Discussion of the example

A number of aspects of the example require further discussion:

(i) Assumptions made in the example include: that no file server accesses are performed (which means that the date program executable code must already be present in the memory of kaka), that the output from *date* is written to the pseudo terminal in a single write, and that *telnetd* passes the output of *date* to *NCSA Telnet* in a single message. The assumptions made highlight the fact that, for any interaction, quite different interaction networks may be observed on repetition of the user action, particularly where workload conditions are substantially different over some set of repetitions.

(ii) In the example, communication is done by various forms of message passing. Communication between *NCSA Telnet* and *telnetd* is by message passing over an ethernet LAN, as is any communication between *date* on kaka and *nfsd* on kiwi.

Communication that involves the pseudo-terminal is by message passing, through the pseudo-terminal input and output buffers, with non-blocking sends and blocking receives. Edges (v_5,v_{10}), (v_{14},v_{24}), and (v_{16},v_{19}) represent communication through the pseudo-terminal's input and output buffers.

Edge (v_{17},v_{13}) represents the time taken to communicate to *csh* the fact that *date* has terminated. This time is likely to be small, but non-zero.

(iii) Interaction models for distributed and multi-processor systems must be able to represent the parallel execution of activities. In distributed systems, threads execute in

parallel over the various nodes of the distributed system and, if a node contains more than one processor, threads execute in parallel within a node. Work for several interactions may be executed in parallel, and, for some interactions, work for a single interaction may be executed in parallel. Any parallel execution within an interaction is obvious from its interaction network.

In the interaction network in Figure 4-6, the writing of the *date* output by *NCSA Telnet*, may occur in parallel with *date* exiting. If kaka were a multiprocessor node then the exit processing of *date* could occur in parallel with *telnetd* passing on the output of *date*. Since kaka is a uniprocessor node then these two activities may not be executed in parallel. Where two or more interactions are executed in parallel, interaction networks might be compared to find any parallelism.

Interactions executing in parallel may be for the same user. For example, parts of the interaction networks for the 'd', 'a', 't', 'e', and 'newline' lexemes may overlap; or, while processing for the 'newline' lexeme is being performed, the user may switch to another *NCSA Telnet* session (which, because it is initiated by the user action of selecting a menu item on the Macintosh, is an interaction in itself), and input lexemes to initiate interactions in the other *NCSA Telnet* session.

(iv) The interaction network for 'newline', and for the other user actions as well, is much more complex than if the interaction had occurred on a centralised system. If the 'newline' interaction had occurred on such a system, then only one node and two threads would be involved. A possible interaction network for that case is shown in Figure 4-7. In the distributed environment, the 'newline' interaction occurred across two nodes and four threads. If file server access(es) are required then three nodes and five processes would be involved.

(v) The fact that each interaction does not normally correspond to the execution of a single program is highlighted by the example. The interaction network includes the partial execution of three programs (*NCSA Telnet*, *telnetd*, and *csh*), and the total execution of one (*date*). Most other approaches to performance measurement of distributed systems are based on the analysis of individual programs (see Subsection 3.5.3).

(vi) Some of the threads in the example are associated with one sub-task for their entire existence (*date*), while other threads are associated with many sub-tasks (*NCSA Telnet*, *telnetd*, and *csh*).

(vi) At the beginning of this Subsection, the possibility was mentioned that *date* on kaka might need to make a file server request, as a result of a page fault perhaps, to an *nfsd* process on kiwi. The file server (kiwi), *nfsd*, and the communication between *date* and

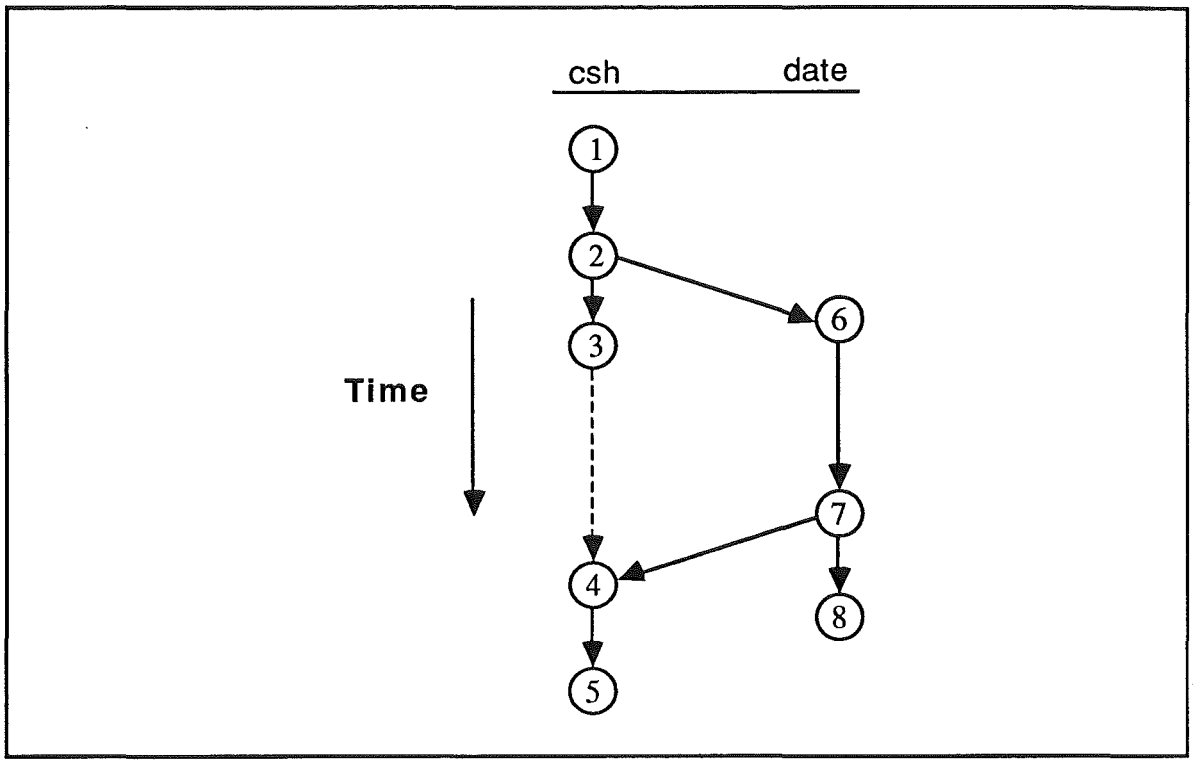


Figure 4-7: Interaction network for 'newline' on a centralised system
(Blocking shown by dashed line.)

nfsd all appeared in the overview in Figure 4-5, but no file server accesses were shown in the interaction network of Figure 4-6 so as to keep the size of the example small.

A file server access is now considered. Assume that a single file server access, to transfer in a page required by *date*, occurs somewhere on edge (v_{11},v_{16}) in the interaction network in Figure 4-6. The extra vertices and edges required to represent the file server access are shown in Figure 4-8. At vertex v_{29}, *date* sends a request message to kiwi asking for the required page, and then blocks waiting for the reply at vertex v_{30}. *nfsd* on kiwi is waiting for request messages, and receives *date*'s request message (vertex v_{32}) as soon as it arrives. *nfsd* retrieves the required page (edge (v_{32},v_{33})), and includes it in a reply message to *date*. *nfsd* then waits for the next incoming request message (vertex v_{34}). After sending the request message, *date* is blocked (edge (v_{30},v_{31})) until the reply is received at vertex v_{31}. *date* then continues as before.

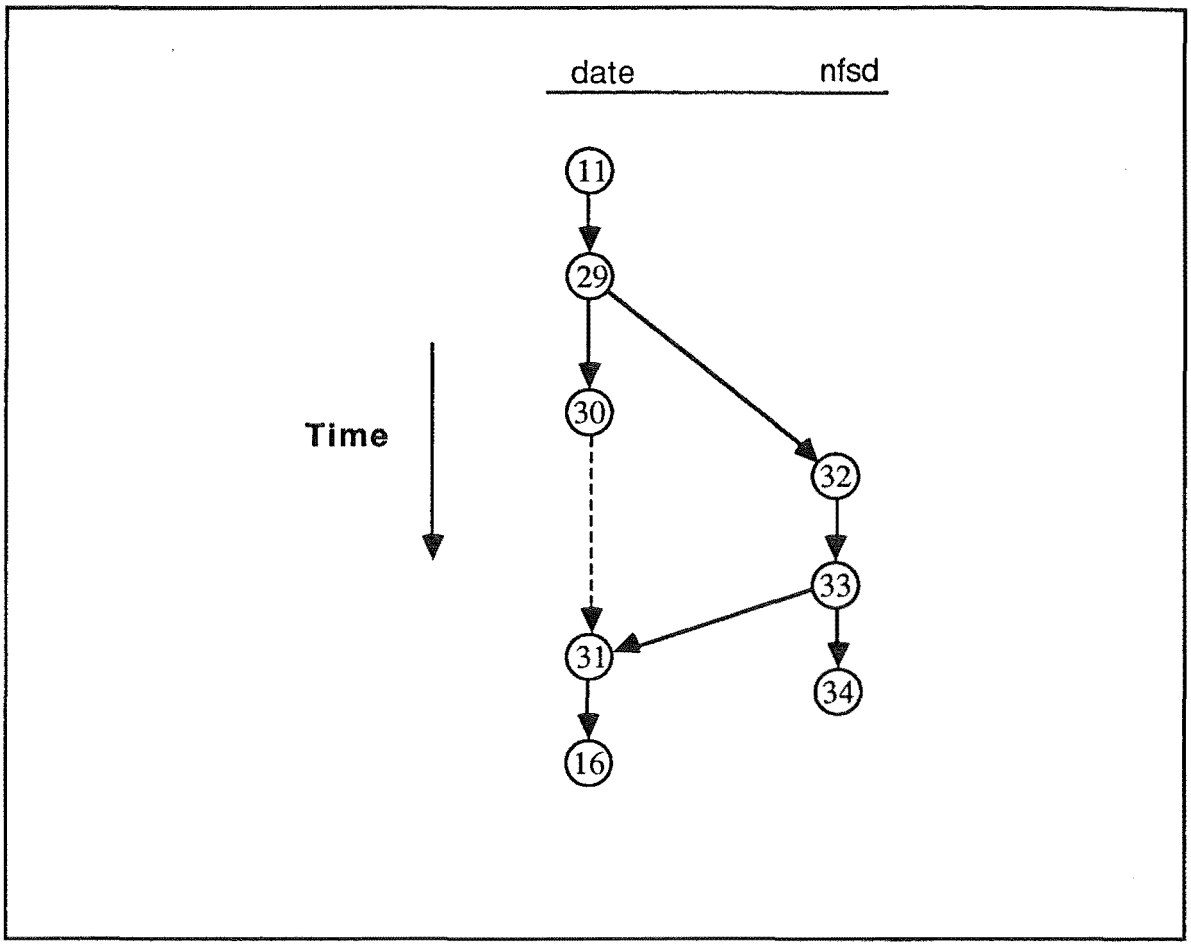


Figure 4-8: Interaction network fragment showing a file server request (Blocking shown by dashed line.)

4.3 Other Representations

In this section, we examine a number of other ways in which the execution of a program has been represented, and compare them with the interaction network. The most important is the *program activity graph* (PAG) ([YANG88], [MILL90a]), which is discussed under (3) below.

(1) Ferrari [FERR83] describes a very simple way of representing the processing involved in servicing an interaction. Ferrari's representation is a time-line showing various events in the processing of an interaction, with some activity occurring between each pair of consecutive events. Activities include periods of use of the processor and I/O devices, and periods of queuing. Because a single time line is used, all of the processing is assumed to be done by a single logical thread. That is, Ferrari's representation does not allow for parallelism or concurrency within an interaction.

(2) Mahjoub [MAHJ84] describes a way of showing how a real-time system performs *service requests*. A service request is generated by an external stimulus. The <external stimulus, service request> interaction model for a real-time system is analogous to our <user action, system reaction> interaction model for an interactive system. In Mahjoub's model, a request is performed by a single logical thread of execution that executes a set of procedures, so the execution of a request can be represented on a time line that shows periods of time spent in each procedure, and periods of time spent waiting for a processor. There is no parallelism within a request, but several requests can execute in parallel.

(3) Yang and Miller describe the program activity graph (PAG): a network representation for the processes and inter-process communication involved in the execution of a distributed program [YANG88], [MILL90a]. The PAG representation for execution of a distributed program is now introduced, with the recording and use of PAGs to be described in later chapters.

A PAG is an acyclic digraph representation of the execution of a distributed program, whose major use is in computation of the critical execution path for a program. Vertices in a PAG correspond to events, and edges to activities. The main events recorded in a PAG are process creation and termination events, and message send and receive events. Activities include all processing between events, which may include execution on a processor, and device accesses. An example of a Program Activity Graph is included in the next chapter as Figure 5-1.

While interaction networks are similar in some ways to Yang and Miller's program activity graphs, there are important differences:

(i) Program activity graphs are concerned with representing the processing of an entire program, whereas interaction networks are concerned with the processing of an interaction. As previously noted, the fact that interaction networks are designed to represent the processing of an interaction is a major difference from most other approaches (including program activity graphs) to performance measurement of distributed systems.

(ii) Only processing performed by processes directly associated with the program is included in a program activity graph, whereas all processing (including that performed by servers) can be included in an interaction network. Figure 4-8 has shown how the processing performed by a file server can be included in an interaction network.

(4) Miller describes the program history graph, an acyclic digraph representation of execution within a *server* program [MILL88a]. A server is defined to be a computation that receives a request from processes outside the computation, computes a result (involving one or more of the processes within the computation), and then returns the result to the requesting process. The vertices of a program history graph represent send and receive

events, with edges connecting successive events within a process, and also connecting the send vertex of a message with the receive vertex of the message.

The main use for a program history graph lies in determining, for each request that a server performs, the message passing that occurred between server processes. Identifying all of the message passing that results from a request is similar to identifying all of the processing that results from a user action, as in both cases the extent of the processing performed by a set of processes as a result of an input (user input or request message) is to be determined.

Although the program history graph is an acyclic digraph representation of processing in a distributed system, its scope (all message passing between the processes of a server) and the events recorded in it (message send and receive only) mean that its uses are very restricted when compared to those of the interaction network.

(5) Tsai *et al* describe a way of representing the execution of a distributed program using IPEL diagrams. The IPEL diagram is an acyclic digraph representation of the execution of a distributed program. Events represented by vertices in an IPEL include process create, terminate, begin execution on processor, and wait; and message send and receive. Edges represent process execution and message passing. The differences between interaction networks and program activity graphs described in (3) are also differences between interaction networks and IPEL diagrams.

4.4 Summary

A user interaction with a distributed system has been defined for this thesis as a <user action, system reaction> pair. The system reaction is seen as a task. The way in which a system performs each task can be described by an interaction network, an acyclic digraph, in which each vertex represents an event in the life of a sub-task and each edge represents an activity performed by a sub-task.

Other work in this area has been described, notably the concept of the program activity graph. It has been shown that there are substantial differences between the concept of the interaction network and the other work described. In particular, the idea of recording the processing done in executing an interaction, rather than the processing in a single program, seems significantly different from other work. This approach of monitoring interactions is very close to the user's view of interactive behaviour, in which the user performs actions and the system responds to them.

The author's concept of the interaction network may be consistent with what Anderson *et al* were advocating in a discussion of future performance measurement tools for Unix systems [ANDR89]. Anderson *et al* say that "Performance methodologies have evolved

considerably over the last two decades from an analysis of system utilization, to a degradation analysis of manageable subcomponents of end-user response time". They go on to say that "A critical requirement for subcomponent analysis of end-user response time is an architected definition of what constitutes the beginning and the ending of a transaction. In the Unix environment this is not the beginning and ending of a process but must be defined from an end-user perspective". If one draws the analogy between a "transaction" and an "interaction", then the interaction network approach does provide a definition of the beginning and ending of an interaction. Also, decomposition of response time into components is an important analysis technique available from recording of interaction networks, as we will show in Chapter 6.

Having presented a model of a distributed system, a model of how users interact with that system, and a way of representing the system response to each user input, we now go on to consider how these ideas can be applied to performance measurement of distributed systems.

Chapter 5

The Critical Path

Of central concern in all studies of interactive performance is the question of how response times can be reduced. We now go on in Chapters 5 and 6 to show how interaction networks can be used for the analysis and improvement of interactive performance. First, we define the *critical path* through an interaction network as a sequence of edges through the graph such that the length of at least one edge must be reduced if response time is to be reduced. Improvement of interactive performance for some set of interactions then becomes a matter of finding strategies that can be expected generally to reduce the lengths of critical paths through interaction networks.

The critical path concept, as used in network analysis, is described in Section 5.1. In Section 5.2, we give an example showing use of the critical path in computer performance evaluation. In the example to be discussed [YANG88], execution of a distributed program is represented by a program activity graph. The critical path through a program activity graph shows those activities whose execution times must be reduced if total execution time of the program is to be improved. Finding the critical path through an interaction network is described in Section 5.3, with critical paths through interaction networks and through program activity graphs being compared in Section 5.4.

5.1 The Critical Path Method

The critical path method (CPM) is widely used in project management to help schedule the various activities that go to make up a project. The activities are represented in what Even calls a *PERT digraph* [EVEN79], an acyclic digraph with the following properties

(1) There is a start vertex s, and a termination vertex t, such that $(t \neq s)$.

(2) Every vertex other than s and t is on some directed path from s to t.

A PERT digraph represents the activities required for a project and the dependences between those activities. Every edge represents an activity and is weighted with the expected duration of the activity. For every vertex, the outgoing edges represent the activities that can be started when all the activities represented by the incoming edges are completed. The outgoing edges of s are the activities that can start at the beginning of a

project, and the incoming edges of t are the activities for which no other activities need wait.

The minimum expected completion time of a project whose activities are represented in a PERT digraph can be found by labelling the vertices of the graph in the following way. Vertex s is labelled with 0. Remaining vertices can be labelled when all of their predecessor vertices have been labelled. Each vertex is labelled with the maximum over all incoming edges of the edge weight, predecessor vertex label sum. The last vertex to be labelled is t, with its label being the minimum completion time for the project represented by the digraph.

The critical path can then be determined by going back from t to s, taking at each vertex the edge which determined the label of the vertex. If there is a choice of edge at any vertex then there is more than one critical path. For a project to complete in the minimum time, each activity on the critical path(s) must be commenced as soon as all of the activities on which it depends have completed, and must be completed in its expected duration. If the project is to be completed in a time less than the calculated minimum, then the duration of a least one of the activities on each critical path must be reduced; reducing the duration of an activity not on a critical path will not reduce the project completion time.

There is some leeway in scheduling an activity not on a critical path in that it is possible to either start it some time after all of the activities on which it depends complete and/or for it to overrun its expected duration, and still to complete the project in the minimum time.

The CPM method is most often used for project planning, where the expected durations of activities and their dependences are estimated. The ideas of representing activities and dependences in a PERT digraph, and of the critical path through the digraph, are useful in computer performance evaluation of parallel and distributed systems, as in these systems computation is performed by many parallel activities.

An example will help to illustrate this point. Consider the interaction network of Figure 4-6. Response time for this interaction can be defined as time(v_{28}) - time(v_1). Some tuning action might be taken to reduce the length of edge (v_{11}, v_{12}). Because of this reduction the event represented by vertex v_{12} occurs earlier, but the event represented by vertex v_{13} occurs no earlier, because edge (v_{12}, v_{13}) represents *csh* blocked waiting for *date* to finish and no action has been taken to make *date* finish any sooner. Therefore, edge (v_{12}, v_{13}) becomes longer by the same amount by which edge (v_{11}, v_{12}) is reduced, and response time remains the same. If, on the other hand, the length of edge (v_{16}, v_{17}) were reduced, then response time would be reduced, as the event represented by vertex v_{13} will occur earlier (edge (v_{12}, v_{13}) can be reduced in length as it represents *csh* waiting for *date* to finish), and so the events represented by vertices v_{14}, v_{15}, and v_{24} to v_{28} would occur earlier.

Traditionally, most computations have been performed by a single logical thread of execution (see for example the description of Ferrari's interaction model in Section 4.3). Because all computation is performed by a single logical thread of execution, response times for such computations are reduced if the duration of any activity in a computation is reduced. The examples above show that where a computation is performed by more than one thread, as it is in distributed and parallel systems, then reducing the durations of some activities will not reduce response time, whereas reducing the durations of other activities will reduce response time. To reduce response time, it is important to be able to identify activities along the critical path. Several authors have noted the importance of finding the critical path for distributed and parallel computations [DUCH89], [GRAY89], [REED89], [YANG88]. Approaches have been developed where some form of PERT digraph is used to represent a distributed computation, from which the set of activities on the critical path can be determined. Two such approaches are now described, with an approach used by Yang and Miller discussed first, followed by a description of our own approach.

5.2 Program Activity Graphs

Yang and Miller [YANG88] have developed an approach to performance evaluation of distributed programs in which the execution of a distributed program is represented as a program activity graph. The critical path through a program activity graph highlights the program activities that should be addressed in order to improve program performance. Program activity graphs can be recorded and analysed by the IPS-2 performance measurement system [HOLL91a].

The example PAG given in [HOLL91a] is reproduced in Figure 5-1. To ensure that a program activity graph is a PERT digraph, *initial node* and *terminal node* are added, with dummy edges of weight 0 from *initial node* to the *Init* vertex of every initial process, and from the *Exit* vertex of every process to *terminal node*. There can be more than one initial process as, when IPS-2 starts a program which it is to monitor, it creates one or more initial processes, as specified by the user's description of the program to IPS-2.

Vertices in a PAG represent events corresponding to the beginning and the end of one or more activities. These events include creation, initiation and termination of processes, and sending and receiving of messages. All of the event types recorded in a PAG are included in the set of events needed in an interaction network to show the sub-task structure, as described in Chapter 4.

Edges in a program activity graph are of two main types. A *process execution* edge is weighted with the amount of time the process performing the activity spent executing

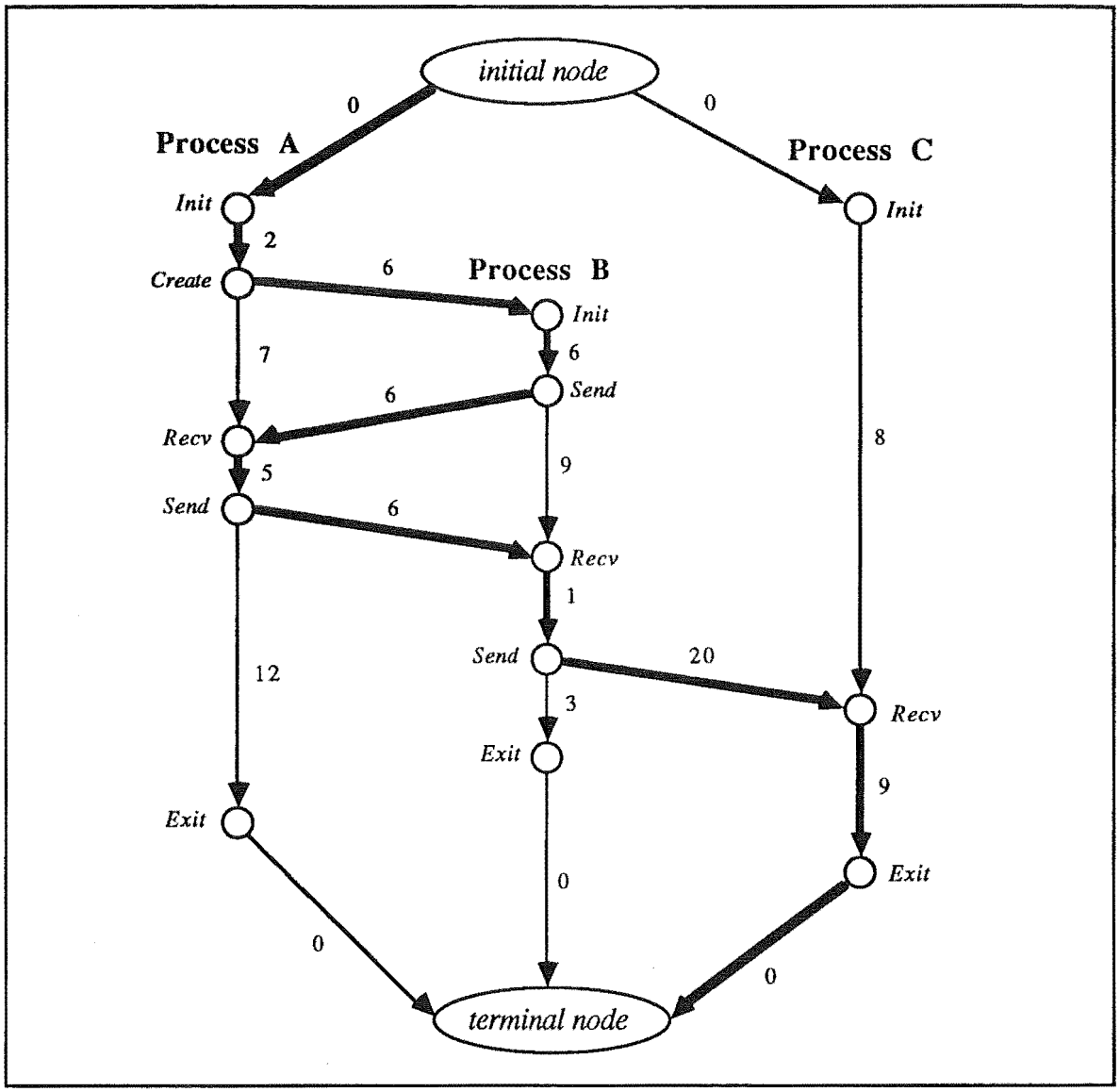


Figure 5-1: Program activity graph example [HOLL91a]

between the two events representing the beginning and the end of the activity. A *message* edge is weighted with a time computed using a formula linear in message length, with the time being an approximation of the minimum message delay. Although estimating message times means that clock synchronisation is not required for recording program activity graphs, such estimates cannot include queuing times and may be much less accurate than measurement where clocks can be closely synchronised.

Each vertex in a PAG represents a dependence, as the activities represented by its outgoing edge(s) can be started as soon as all of the activities represented by its incoming edge(s) are complete. Consider, for example, a *recv* vertex in Figure 5-1. It has two incoming edges (a process execution activity and a message activity) and one outgoing edge

(a process execution activity). The outgoing process execution can begin as soon as **both** the process execution activity of the incoming process-related edge has finished and the message has arrived.

The critical path through a PAG is also the longest path, making it straightforward to find. In Figure 5-1, the critical path through the PAG consists of the thick edges.

5.3 Interaction Networks

We now discuss the critical path (or critical paths, as there may be more than one) through an interaction network. First, in Subsection 5.3.1, we give an algorithm for finding the critical path through an interaction, and show:

(1) That there is at least one critical path through every interaction network.

(2) The conditions under which an interaction network has more than one critical path, and the conditions under which an interaction network has exactly one critical path.

In Subsection 5.3.2, we describe how weights are assigned to the edges of an interaction network, and give some results on the lengths of paths through an interaction network. In particular, we show that all critical paths through an interaction network must be of equal length, and that response time can be reduced if and only if the length of at least one edge on each critical path through an interaction network is reduced.

In Subsection 5.3.3, the use of critical path information in performance evaluation, and some potential problems that arise, are discussed. Finally, in Subsection 5.3.4, a situation is described where the final vertex in a critical path might be chosen in a way different to that described in Subsection 5.3.1.

5.3.1 Defining the Critical Path

An interaction network differs from a PERT digraph in that interaction networks can have more than one sink vertex. Corresponding to the last event in an interaction, there will be a sink vertex v_f, such that $time(v_f) \geq time(v_i)$ for all vertices v_i in an interaction network. In theory at least, there may be two or more "last events" that occur at precisely the same instant. Since it makes no difference to the arguments that follow in this subsection, we will refer to all of them as v_f.

We define *response time* for an interaction to be $time(v_f) - time(v_0)$, where vertex v_0 is the source vertex in an interaction network. A critical path through an interaction network runs from v_0 to a v_f vertex, and consists of a set of edges such that to reduce the length of the critical path (and therefore to reduce response time), the length of at least one edge in the set must be reduced. A list of the edges on the critical path, and of the activities that

they represent, is an important piece of performance information, as to reduce response time for an interaction the length must be reduced of at least one of the edges on each critical path.

Before giving our definition of the critical path through an interaction network, we first define the B-edge. A *B-edge* is an edge that represents a period during which a sub-task is blocked waiting to synchronise with some other sub-task within the same task. To represent B-edges in an interaction network, it is necessary to record events additional to those identified in Chapter 4. The additional events are those that signify that a sub-task has blocked waiting to synchronise with another sub-task. These events include a process waiting to receive a message, a message being put into a queue to await delivery to a process, and a process waiting for another process, for example a parent process waiting for one of its children to terminate.

A *critical path* through an interaction network is then defined as any path from vertex v_0 to a v_f vertex that contains no B-edges.

Before considering some theorems on the nature of critical paths through interaction networks, some further discussion of B-edges is required. Each B-edge corresponds to a period in which a sub-task is waiting for another sub-task in the same task. The only vertex types that can represent the end of a period during which one sub-task waited for another are those where an outgoing edge is dependent on more than one incoming edge, that is the join and message join/split vertex types.

(1) Join vertices. A join vertex represents the joining of two sub-tasks into one. In the most common case, one sub-task, S_A becomes ready to join before the other. The edge associated with S_A that is incident on a join vertex is a B-edge, as the edge represents a sub-task waiting to join with another sub-task in the same task. The other edge incident on such a join vertex cannot be a B-edge, because if it were then there would be two sub-tasks ready to join but with each waiting for the other. Very occasionally, two sub-tasks may become ready to join at exactly the same instant, but most commonly each join vertex will have one incident B-edge.

(2) Message join/split vertices. A message join/split vertex has N incoming edges. The N-1 incoming edges that represent the first N-1 sub-tasks (in the order that the associated message data arrived in the buffer) are all B-edges, as these edges represent the period for which each message fragment spend waits for the Nth and final fragment to arrive.

We can now present three important theorems about critical paths.

Theorem 1: Every interaction network has at least one critical path from vertex v_0 to each v_f vertex

Proof: We must show that from vertex v_0 to each v_f vertex, there is always at least one path that contains no B-edges. An algorithm that identifies such a path in reverse, from a v_f vertex to vertex v_0, is given in Figure 5-2. There are no B-edges on the path traversed because each vertex that has an incoming B-edge also has at least one incoming non-B-edge that can be followed in reverse. Also, the algorithm must eventually reach vertex v_0 as it is the only source vertex in an acyclic digraph.

```
Set the current vertex to a vf vertex.
Repeat
        If the current vertex is a join vertex, set the current vertex to a predecessor
                connected by a non B-edge,
        else
                If the current vertex is a message join/split vertex, set the current vertex to
                        the only predecessor connected by a non B-edge,
                else
                        Set the current vertex to the only predecessor.
Until the current vertex is v0.
The edges traversed are a critical path.
```

Figure 5-2: An algorithm for finding a critical path through an interaction network.

Let us consider use of the algorithm given in Figure 5-2 to find the critical path through the interaction network shown in Figure 4-6. In the figure, v_0 is vertex v_1 and v_f is vertex v_{28}. The algorithm starts at vertex v_{28}. The only vertex where there is a choice of paths to follow is vertex v_{13}. Edge (v_{12}, v_{13}) represents *csh* waiting for *date* to exit, so the critical path does not include edge (v_{12}, v_{13}). The critical path through this interaction network, then, passes through vertices v_1, v_2, v_4, v_5, v_{10}, v_{11}, v_{16}, v_{17}, v_{13}, v_{14}, v_{24}, v_{25}, v_{27}, and v_{28}. The critical path found illustrates discussion at the end of Section 5.1 of the effects on response time of possible reductions in the durations of certain activities.

Theorem (2): An interaction network may have more than one critical path.

Proof: An interaction network has more than one critical path if either (or both) of the following is true.

(i) Both edges incident on some join vertex that is on a critical path are non-B-edges. One critical path cannot contain both edges incident on a join vertex, as a path through an acyclic digraph cannot pass through a vertex more than once. Therefore, to show that more than one critical path exists, it is sufficient to show that each of these two non-B-edges is on a critical path. Each of the two edges is on a critical path because:

(a) the join vertex is on a critical path, so there is a path from the join vertex to a v_f vertex that contains no B-edges.

(b) each edge incident on the join vertex is a non-B-edge, so there is a path from each predecessor of the join vertex to a v_f vertex that contains no B-edges.

(c) using the same argument as used in the proof of (1), there must be a path that contains no B-edges from vertex v_0 to each of the predecessors of the join vertex. In fact there is such a path from vertex v_0 to every vertex in an interaction network.

From (b) and (c) it is clear that a critical path, a path containing no B-edges, passes through both of the edges incident on the join vertex, and therefore there are at least two critical paths through an interaction network that has a join vertex on the critical path with two incident non-B-edges.

(ii) There is more than one v_f vertex. The algorithm given in Figure 5-2 can be used to find a critical path from each v_f vertex to vertex v_0. Each of these critical paths is different, as each ends at a different vertex.

Theorem (3): If an interaction network has a single v_f vertex, and there is no join vertex with two incident non-B-edges on a critical path, then there is a single critical path through the network.

Proof: Consider running the algorithm given in Figure 5-2 on an interaction network of the type described in the statement of the theorem. First, the algorithm must start at the only v_f vertex in the interaction network. Second, every vertex on a critical path has only one non-B-edge incident on it, so at each vertex considered by the algorithm there is only one choice as to which edge to follow. Therefore, there is only one critical path from vertex v_0 to the only v_f vertex for this class of interaction networks.

5.3.2 Edge Weightings

We now describe how the weighting of each edge in an interaction network is determined, and then give two further theorems. Each edge in an interaction network is weighted by the elapsed time of the activity it represents, that is edge (v_i, v_j) is weighted with time(v_j) - time(v_i). To determine accurately the elapsed time of an activity that begins on one node and ends on another, such as the sending of a message across a network, the

clocks of all nodes must be closely synchronised. The important issue of clock synchronisation will be dealt with in Chapter 8.

Every edge is weighted with the elapsed time of the activity that it represents. The length of a path through an interaction network is the sum of the weights on the edges that make up the path. Therefore, all paths from vertex v_i to vertex v_j are of equal length, because the sum of the elapsed times of a path of activities from vertex v_i to vertex v_j must be equal to the elapsed time between the two events, that is time(v_j) - time(v_i).

In particular, all paths from vertex v_0 to a v_f vertex must have the same length. Furthermore, if more than two or more v_f vertices exist, then by definition they all represent events that occurred at exactly the same time. Therefore, all critical paths through an interaction network have the same length, and that length is equal to response time.

Theorem (4): Response time can be reduced if and only if the length is reduced of at least one edge on each critical path through the interaction network.

Proof: Three cases must be considered:

(i) If the length of an edge not on a critical path is reduced, then the length of the critical path is not reduced, and therefore response time is not reduced.

(ii) If the lengths are reduced of edges on some, but not all, of the critical paths, then the number of critical paths is reduced but as at least one critical path of the original length remains, response time is not reduced .

(iii) If the lengths are reduced of one or more edges on all critical paths then, as the total length of each of the critical paths has been reduced, response time is reduced.

The proof assumes that all dependences between activities are shown explicitly in an interaction network. If this is not the case then, for example, some activity not on the critical path might be reduced in length, which might cause (through some unrecorded dependence) an activity on the critical path to be reduced in length, which in turn reduces the length of the critical path. This issue is discussed further in Subsection 10.4.1 in relation to dependences that arise from the sharing of resources.

Theorem (5): If each B-edge is assigned a weight of zero, then every critical path through an interaction network is a longest path through the network, and every non-critical path is shorter than the longest path.

Proof: If each B-edge is weighted with the elapsed time of the activity that it represents, then all paths from vertex v_0 to a v_f vertex are of equal length. If we assign instead a weight of zero to each B-edge, the length of every critical path is not affected, as a critical path contains no B-edges. All other paths from vertex v_0 to a v_f vertex are reduced in length.

If a weight of zero is assigned to each B-edge then each critical path through an interaction network is a longest path. This is consistent with critical path analysis of PERT digraphs and program activity graphs, where each critical path is a longest path. In the remainder of this thesis, however, we will weight B-edges with elapsed activity times and critical path analysis will be based on identifying of B-edges.

5.3.3 Use of the Critical Path in Performance Evaluation

We now show below how analysis of critical paths can be used for performance improvement. In (1), it is assumed that there is a single critical path through every interaction network. Interaction networks with more than one critical path, and interaction networks with near-critical paths (paths likely to become critical paths) are discussed in (2).

(1) Analysis of critical paths in performance evaluation

An obvious way in which analysis of critical paths can be used in performance improvement is to record details of all activities, and to produce a summary of all activities on the critical path. Summaries could be prepared for individual interactions, or for arbitrarily chosen sets of interactions. For a set of interactions a summary may be based on total time spent performing each type of activity along the critical paths of all interactions in the set. Tuning actions could then be taken to reduce the duration of one or more activities on the critical path(s), with the aim of reducing response times. Questions of which activities to record and how to record them are addressed in Chapter 6.

(2) Near-critical paths

Although reducing the duration of one or more activities on the critical path will result in some reduction in response time, the reduction achieved in practice may be small because of the presence of near-critical paths in an interaction network. We now:

(i) Describe near-critical paths, and show that the composition of any near-critical path is, for purposes of performance evaluation, nearly as important as the composition of a critical path.

(ii) Show that the case of more than one critical path through an interaction network can be treated the same as the case of one critical path and several near-critical paths.

(iii) Describe performance evaluation of interaction networks that contain near-critical paths.

The effects of a near-critical path on reduction of response time is illustrated by the two interaction network subgraphs in Figure 5-3. Subgraph (a) represents part of a task before a tuning action is taken, and subgraph (b) represents the same part of the task after that

tuning action. Assume that, in (a), the critical path runs through vertices v_1 and v_2. Some tuning action is taken that will reduce the length of edge (v_1,v_2), an edge on the critical path, as shown in (b). In (b), the critical path runs through vertices v_1, v_3, and v_4. Although the length of an edge on the critical path, edge (v_1,v_2), has been substantially reduced, the length of the critical path has been reduced by a much smaller amount.

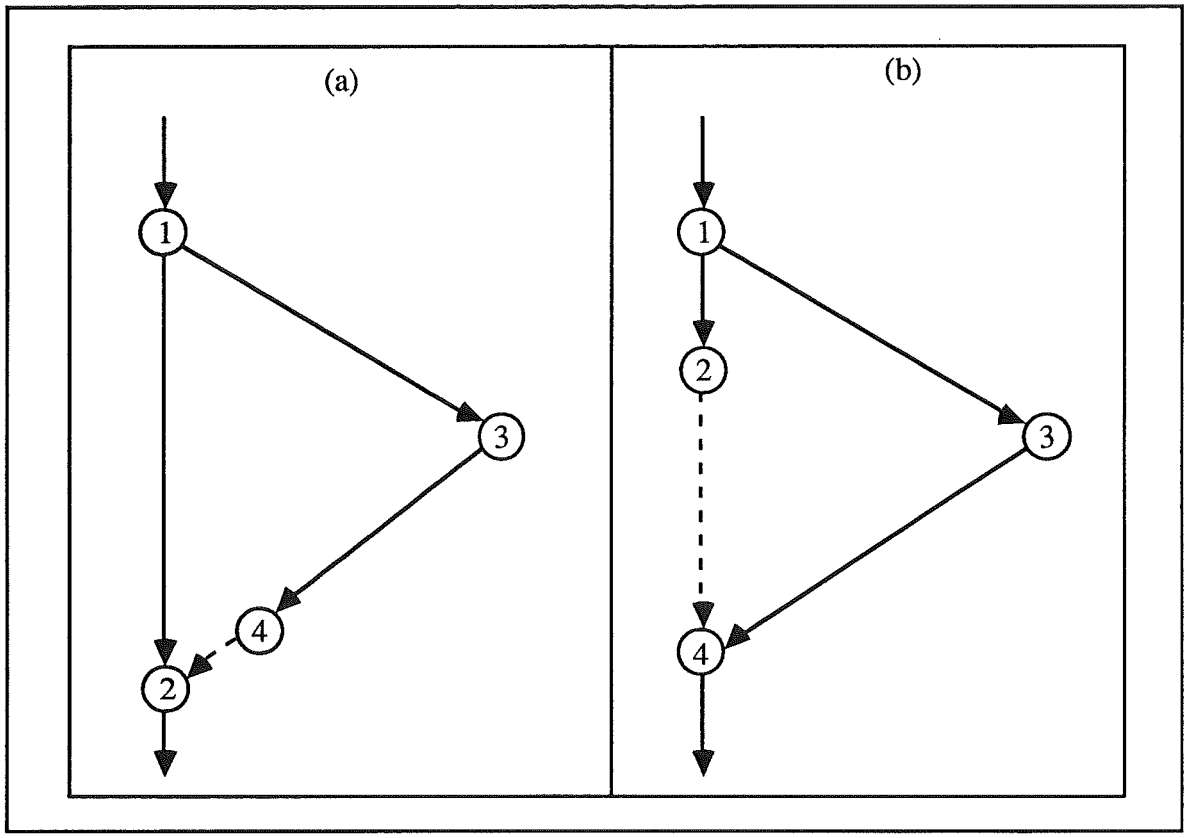


Figure 5-3: Critical and near-critical paths (B-edges shown as dashed lines; the vertical dimension is linear in time)

The path through vertices v_1, v_3, v_4, and v_2 in Figure 5-3(a) is an example of what we shall call a near-critical path. A *near-critical* path is a path to whose length B-edges make only a small contribution. A near-critical path is likely to become a critical path if some action is taken to reduce the length of the original critical path(s). Anderson and Lazowska have commented on the existence of near-critical paths through a program activity graph [ANDE90].

In Figure 5-3(a), there is a near-critical path, through vertices v_1, v_3, v_4, and v_2, with one short B-edge, edge (v_4, v_2). In Figure 5-3(b), a tuning action has reduced the length of edge (v_1,v_2), and has caused the event represented by vertex v_2 to occur before the event represented by vertex v_4, thereby eliminating the B-edge (v_4, v_2) in (a), and introducing the

B-edge (v_2,v_4). Because a different sub-task arrives at the join first, the B-edge has "flipped" from one joining sub-task to the other.

Another slightly different example is that case where a sink vertex v_s exists such that time(v_s) is very close to time(v_f). After some tuning action, vertex v_s could readily become the final vertex in the interaction network. The path from v_0 to v_s is therefore also a near-critical path.

For an interaction network with more than one critical path, we can arbitrarily choose one critical path to be "the" critical path, and regard all other critical paths as near-critical paths. The existence of near-critical paths means that it is not enough to deal simply with critical paths. Miller *et al* recognised that near-critical paths needed to be accounted for in the analysis of critical paths through program activity graphs [MILL90a]. They refer to near-critical paths as being the second-longest critical path, the third-longest critical path, and so on.

For analysis of individual interaction networks, near-critical paths can be allowed for in a number of ways:

(i) Statistics on any near-critical paths can be presented alongside those of the critical path. Near-critical paths through a program activity graph can be found and ordered accorded to decreasing path length. Near-critical paths through an interaction network can also be found, and ordered by the contribution made by the sum of lengths of non-B-edges along each path. Alternatively, activity summaries might be based on the total durations of activities on all critical paths and near-critical paths, averaged over the total number of both critical and near-critical paths.

(ii) It is possible to perform "what-if" analysis on a program activity graph or interaction network. The weights of one or more edges can be changed to simulate the effects of possible tuning actions, and the effect on the critical path observed. Any switch to a near-critical path should be obvious.

(iii) Miller *et al* found that, in their experience, near-critical paths share most of their edges with the critical path [MILL90a]. Even where that is not true, the compositions of the critical path and a near-critical path may be similar in terms of the activities performed. In either case, a tuning action taken to reduce the length of a critical path would have almost as much effect in shortening the length of a near-critical path, so the presence of near-critical paths might be ignored in these cases (which common sense suggests should be the majority of cases).

One of our principal concerns is, as will be discussed later, to pool components of critical paths for large sets of interactions. Provided that there are only infrequent cases where a critical path and near-critical path are very different in terms of the activities

performed, a strategy aimed at a general reduction in lengths of critical paths should be effective. Techniques suggested in (i) and (ii) could also be used in analysing critical paths for sets of interactions, particularly the averaging technique suggested in (i).

5.3.4 Other Choices for the Final Vertex

In some situations, the response time with which the user is concerned may be considerably less than the duration of the system reaction [SOMI91]. As an extreme case, consider the situation where a user enters a command that initiates a long-running task in the background. The user may be more concerned with the time taken before the next command can be entered, rather than the time taken for the background task to finish.

To deal with such situations, software for interaction network analysis could determine the event that signifies the end of the period of response time for which the user is delayed, and then find the critical path between vertex v_0 and the vertex that represents this event. This event will usually be a final event, but will sometimes be an earlier event such as "user prompted for next input". Another possibility is to allow a performance analyst to select, during analysis of an individual interaction, the vertex at which response time (and therefore the critical path) finishes.

5.4 Comparisons

We now compare the concepts of the critical path through an interaction network and the critical path through a program activity graph [HOLL91a], [MILL90a], [YANG88]. Although there are similarities, there are also significant differences. First, differences between the algorithms for determining the critical path are discussed, then important conceptual differences are described.

(1) Comparison of the complexity of the critical path algorithms

The critical path through a PERT digraph and a program activity graph is the longest path, and algorithms that find the longest path through this type of digraph take O(|edges|) time. In the case of an interaction network, the "labelling" step of the CPM critical path algorithm has effectively already been done, as each vertex is labelled with the time at which the event it represents occurred. The algorithm for finding the critical path through an interaction network takes advantage of this fact, so that all it needs to do is to follow a critical path back from a v_f vertex to v_0. This algorithm therefore takes O(|edges on the critical path|) time.

(2) Conceptual differences

A program activity graph represents execution of a program in terms of activities and dependences between activities. Activities are execution on a processor and message passing, with dependences being synchronisation of processes with other processes or with messages. Each edge of a program activity graph is weighted with the "execution time" of the activity represented: edges that represent thread activity are weighted with the processor time used during the activity, and edges that represent message passing are weighted with expected network delay as determined by a formula linear in message length [YANG88]. Queuing times are not included in edge weightings.

The critical path through a program activity graph is defined to be the longest path through the graph. Yang and Miller comment that "the critical path information in our discussion ignores the delays caused by competition for the external resources [such] as CPUs, so that it depends entirely on the structure of the program" [YANG88]. In other words, they are trying to capture in a program activity graph the structure of a program in terms of activities and dependences irrespective of workload, and to weight edges in the graph with the "execution" time for the activity, rather than the elapsed time. A program activity graph is therefore very similar to a PERT digraph used in CPM: both the program activity graph and the PERT digraph include representations of activity durations and dependences and, in both, the length of the critical path is the **minimum** time for completion of the project or program, rather than the completion time for any particular "execution" of the project or program.

In summary, a critical path in a CPM graph shows the activities expected to be the most critical based on predictions. A critical path in a program activity graph shows the activities expected to be the most critical based on measurement of a program execution. For a program activity graph, Yang and Miller attempt to eliminate the effects of competing workload by, for example, ignoring queuing times.

An interaction network, however, is designed to record details for one particular execution of an interaction. Because the competing workload has a very significant effect on response time, the effects of workload are reflected in an interaction network. Examples of this include:

(i) Each edge weight includes any times during an activity that were spent waiting for a resource to become available.

(ii) Every access to a file server that was performed to service a page fault is recorded in an interaction network. The number of page faults that occur during a task depends primarily on the competition for physical memory during the execution of the task.

The same interaction performed under widely varying workload conditions will usually have very different interaction networks recorded for it, each with a quite different critical path. In contrast, Yang and Miller try to ensure that the same program activity graph is recorded for all program executions, regardless of the workload conditions.

The interaction network is therefore fundamentally different from a program activity graph. A program activity graph is designed to be used to improve program performance. It therefore provides a representation of the execution of a program designed to eliminate the effects of a competing workload. An interaction network, however, is intended to be useful in tuning, so the effects of competing workload are clearly visible in an interaction network. In addition, it will be shown in Chapter 6 that analysis of critical paths through interaction networks can be used to derive performance information for sets of interactions that make up all or part of a system workload. Program activity graphs are used only for the analysis of individual programs.

5.5 Summary

Identification of the critical path is important for the analysis of both interaction networks and program activity graphs, but the information given by the critical path through each is quite different. Program activity graphs are designed to eliminate effects of a competing workload to allow the performance analyst to concentrate on a program in isolation. Interaction networks, however, are designed to show also the effects of prevailing workload, as response times to users are heavily influenced by workload. Techniques will be described in Chapter 6 for augmenting the interaction network with performance-relevant events that enable the activities on the critical path to be determined.

Performance evaluation techniques making use of interaction networks will be described in Chapter 6. While some of these make use of information about critical and near-critical paths, analysis of critical paths is not the only performance evaluation technique used.

Chapter 6

Performance Analysis

We now describe methods by which performance information can be derived from interaction networks. Analysis of critical paths through interaction networks is an important part of several of these methods.

To provide useful performance information, further event types must be recorded in interaction networks, as will be explained in Section 6.1. A major goal in recording further event types is to enable the state of a sub-task to be determined during each activity. A way is introduced of defining the state of an activity in terms of the system objects in use during that activity. The event types are described that must be recorded to provide the information necessary to determine the state of an activity. By the end of Section 6.1, we will have defined all event types that must be recorded in an interaction network to support the performance analysis methods described in the remainder of the chapter.

A performance analyst may be interested in analysing a single interaction at a time. Analysis of the network for a single interaction would be done to explain causes of poor response time for an important type of interaction, such as the compilation of a particular C program. Alternatively, a performance analyst may be interested in evaluating performance for some set of interactions, with the aim of improving overall system performance. The ways in which interaction networks can be used to provide performance information to support these two types of analysis are described in Sections 6.2 and 6.3. Finally, in Section 6.4, the range of performance information that can be obtained from interaction networks is compared with that available from other monitors for distributed systems.

6.1 Performance-relevant Events

Some event types that should be recorded in an interaction network have been described in Chapters 4 and 5, and these are summarised briefly in Subsection 6.1.1. An interaction network containing only events of those types can yield useful performance information, as illustrated by some examples in that subsection. Much more performance information can, however, be derived from an interaction network if it contains events of other types, referred to in Chapter 4 as performance-relevant events. The event types introduced in Chapter 5 are one group of performance-relevant events, but the question remains as to

what other event types might be recorded in an interaction network to provide useful performance information.

In Subsection 6.1.2, we show that the ability to determine the state of each activity of each sub-task is important for performance analysis using interaction networks. In that subsection, we describe a way in which activity states can be defined, and we introduce additional event types that provide information on state changes. Changes of state that occur at each event are described in Subsection 6.1.3, and it is concluded that it is possible to maintain states for all sub-tasks in an interaction network. Finally, the issue of the selection of events to be recorded during a particular measurement session is covered in Subsection 6.1.4.

6.1.1 Recapitulation

In Chapter 4, we identified the structural event types, which are the types of event that must be recorded in an interaction network to enable the sub-task structure of a task to be represented. Events of these types include: the beginning and ending of periods during which a sub-task is associated with a thread; the sending and receiving of messages by threads; the reading and writing of shared variables by threads; and events related to thread synchronisation. All event types represented by source, sink, fork, join, message join/split, and multi-way fork vertices are types of structural event, as each represents a change in the number of sub-tasks. Also, some structural events are represented by simple vertices, some message receive events for example, as are all other event types included in an interaction network.

In Chapter 5, further event types were introduced. These types correspond to events that represent the beginning of an activity during which a sub-task is waiting to synchronise with another sub-task, and they are recorded in an interaction network so that its critical path can be determined. Events in this class include: a thread blocking and waiting to synchronise; and the addition of a message to a queue of messages waiting to be received.

An interaction network containing only events of the types described in Chapters 4 and 5 can yield useful performance information. For example, the various nodes and threads that performed part of a task can be identified. Statistics can be calculated that describe the message passing that occurred during a task. The critical path can be identified, and the contributions to the critical path made by each thread and by passing messages between each pair of threads can be calculated. With the addition of further event types, an interaction network can, however, provide much more performance information.

6.1.2 Events Needed to Determine States

The overall aims and objectives for recording of interaction networks should be used to guide decisions on further types of event to be recorded in an interaction network. One of the most useful types of performance information for interactive systems is a decomposition of the response time of an interaction into the times spent in some set of states (as described in Section 3.4). This information helps in finding out how the (response) time of an interaction was spent.

In an interaction network, each edge represents a period of time spent by a sub-task in a single state. In earlier work [PENN88] on interactions in which system reactions are performed by a single logical thread, we defined states in terms of the resources that were currently in use by, or being queued for by, the logical thread. In that work, each state corresponded to use of some combination of resources and resource queues, and was described by the set of resources in use, and the set of resources being queued for.

A generalisation of this model, to be described in (1) below, can be used in defining states of sub-tasks. The additional types of event required to allow states to be determined are discussed in (2). Finally, we show in (3) that the framework established in (1) and (2) provides a very general way of defining states of sub-tasks.

(1) The object-use framework

The state of a sub-task during an activity can be defined by the set of objects used during that activity. We call this method of defining the state of a sub-task the *object-use framework.* An object can be anything whose use might contribute to the time needed to complete a system reaction. Possible classes of objects include: hardware objects, such as processors and discs; software objects, such as procedures, semaphores, and queues; and data objects, such as files, and data elements in a database.

The state of a sub-task during an activity is the state of the thread or the message that the sub-task is associated with during that activity. The state of a thread is given by the set of objects in use by the thread; objects often used by threads include: processors, discs, procedures, and files. The state of a message during an activity is given by the set of objects in use by the message; objects often used by messages include: message queues and segments of communication networks.

The state of a task at some instant is the union of the states of all of the sub-tasks that belong to the task at that time. The state of task can be represented as a set of <sub-task identifier, sub-task state> pairs. The state of an edge in an interaction network is the state of the activity that the edge represents. Hereafter we refer to "the state of edge (v_i, v_j)", because it is less cumbersome than "the state of the activity represented by edge (v_i, v_j)".

Provided that **all** changes in object use are recorded as events, then the state of a thread or message does not change between successive events.

(2) Object-use events

To determine the state of each activity in a sub-task, we must record when a sub-task begins using each object (so that the object can be added to the sub-task state), and when a sub-task stops using that object (so that the object can be removed from the sub-task state). We call these events *begin_use* events and *end_use* events respectively. If the begin_use or end_use of an object is not represented by one of the event types described in Chapters 4 and 5, then a simple event is recorded in an interaction network to represent both the event and the change in sub-task state. In fact, the event types described in Chapter 5 are all begin_use events, where the object in question is a queue in which a sub-task can wait for another sub-task.

If sub-task states are to be determined, an interaction network must contain structural events, and begin_use and end_use events. All of the analysis techniques described in Sections 6.2 and 6.3 are based on recording events of these types. Each change in the state of a sub-task state occurs as the result of an event, so the way in which each event changes sub-task states must be defined. These definitions are given in Subsection 6.1.3.

(3) Generality of the object-use framework

This concept of object-use can be applied for a very wide range of system resources. Groups of objects whose use might be recorded in an interaction network include:

(i) hardware objects, such as processors, memory, and I/O devices. For example, when a processor P is allocated to a thread an event `begin_use(P)` is recorded, and when P finishes a period of execution of a thread an event `end_use(P)` is recorded.

(ii) software objects, such as modules, queues, and locks. For example, when a module M is called an event `begin_use(M)` is recorded, and when M returns an event `end_use(M)` is recorded. Also, when a thread or a message is placed on a queue Q an event `begin_use(Q)` is recorded, and when the thread or message leaves Q an event `end_use(Q)` is recorded.

(iii) data objects, such as database records and files. For example, when a thread opens file F an event `begin_use(F)` is recorded, and when the file is closed an event `end_use(F)` is recorded.

Often, the end_use of one object will coincide with the begin_use of another, a common example being the end_use of a queue object and the begin_use of the object that

members of the queue are waiting to use. In such cases only one event need be physically recorded.

This common treatment of objects as resources of any type, has a parallel in approaches to protection that are based on definitions of an access matrix (see, for example, [SILB91]). In an access matrix, the protection of each system object is described in a column of the matrix. System objects are anything in the system for which protection information must be stored, including hardware objects (such as processors and discs) and software objects (such as files, processes, and semaphores).

Some objects define the state of a sub-task at the operating system level. We call these objects the *system-level objects.* The *system-level state* of a sub-task activity is defined by the set of system-level objects that are in use during the activity. At the system-level, the traditional first level of decomposition for the states of a thread [SILB91] are: running (using a processor), ready (on a processor queue), or waiting (queued for something other than a processor). A message may be: in transit (on a communication network), or queuing (in some message queue).

In the object-use framework, the system-level state of each thread can be recorded in an interaction network by defining as objects: each processor, each processor queue, and each system queue. For many shared resources, such as a disc, two objects can be distinguished: a disc object that services requests, and a queue object that contains requests waiting to access the disc. A similar set of objects can be listed for messages, with each segment of a communication network and each message queue defined as an object. Times spent in the system-level states can therefore be recorded using the object-use framework.

The object-use model is sufficiently general that none of the performance analysis methods we will present in this chapter requires any events other than structural and object-use events to be recorded in an interaction network.

6.1.3 Events and State Changes

If the set of objects currently in use by a sub-task is to be used to define its state, we must specify how each event changes a sub-task's state. In an interaction network, the begin_use and end_use events introduced earlier in the chapter are represented by simple vertices. When a `begin_use(Obj)` occurs in a sub-task, then Object `Obj` is added to the set of objects that define the sub-task's state. When an event `end_use(Obj)` occurs, Object `Obj` is removed from the set of objects that define the sub-task's state. As mentioned above, the event types described in Chapter 5 are all begin_use events, and their effect on sub-task state is the same as that of any other begin_use event.

The effects on sub-task state of the structural event types described in Chapter 4 are discussed in the remainder of this subsection. Five groups of structural events can be identified, as events involving: a thread sub-task; a message sub-task; a thread sub-task and a message sub-task; two thread sub-tasks; and two or more message sub-tasks. Each of these groups is discussed below, with a summary concluding the subsection.

(1) A thread sub-task

Structural events involving a single thread sub-task are represented by source vertices and sink vertices. At a source vertex, the current state of the thread that detects the user action must be established, so that the state of the sub-task edge leaving the source vertex can be correctly determined. A sink vertex represents the end of a thread sub-task.

(2) A message sub-task

Structural events involving a single message sub-task are represented by source vertices and sink vertices. A message source vertex indicates that the user action caused a message to be created, with the source vertex representing the creation of this message. The message contains the data that was input by the user. As the source vertex represents the creation of a message, the state of the sub-task edge leaving the source vertex is the state in which the message was created. A sink vertex represents the end of a message sub-task.

(3) A thread sub-task and a message sub-task

Events that involve a thread sub-task and a message sub-task are message send and receive events. A message send event is represented by a fork vertex, as shown in Figure 6-1(a). The state of the outgoing edge that corresponds to the message is determined by the state in which the message is created, with a message usually either placed into a queue (queue object in use), or directly onto a communication network (network segment object in use). The state of the outgoing edge that corresponds to the thread is the result of a simple transformation on the state of the incoming edge. Often, the state of that outgoing edge will be the same as that of the incoming edge, as the act of sending a message does not usually change the objects being used by a thread. In a multi-cast send operation, there are one or more outgoing edges that represent message sub-tasks, with the state of each edge determined by the state in which the message it represents is created.

Some message receive events are represented by join vertices, while others are represented by simple vertices (see Subsection 4.2.4). An example of a message receive event represented by a join vertex is shown in Figure 6-1(b). The state of the outgoing edge is the result of a simple transformation on the state of the incoming edge that corresponds to the thread. Often, the state of the outgoing edge will be the same as that of

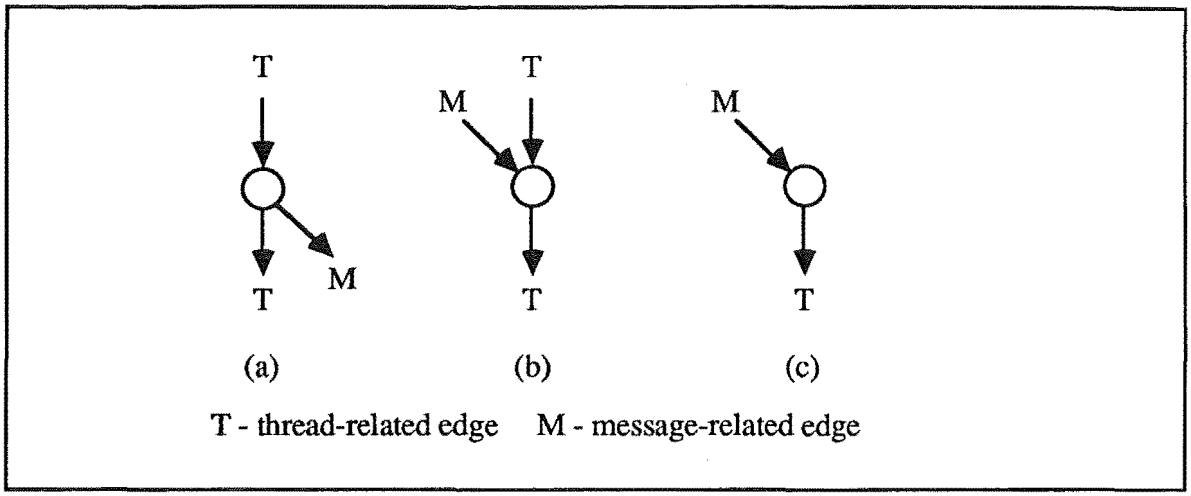


Figure 6-1: Vertices that represent the interaction of thread and message sub-tasks

the incoming edge that corresponds to the thread. Another common transformation will be for the state of the incoming edge that corresponds to the thread to include use of a queue object, and for the state of the outgoing edge to exclude use of the queue object, and to include use of a processor queue object. The state of the incoming edge that corresponds to the message may or may not influence the state of the outgoing edge.

An example of a message receive event represented by a simple vertex is shown in Figure 6-1(c). Here, the state of the outgoing edge is the result of a simple transformation on the state of the thread prior to the reception of the message. The problem of determining this state is similar to that of determining the state of the outgoing edge for source vertices that are thread-related.

(4) Two thread sub-tasks

Events relating to two threads occur in two situations. In the first, one thread (the parent) creates a second thread (the child), with this event represented by a fork vertex. The state of each of the outgoing edges depends on the implementation of the thread creation operation. Often, the parent and the child share many objects, such as file objects and memory objects, so those objects will appear in the states of both the outgoing edge that corresponds to the parent thread and the outgoing edge that corresponds to the child thread. Other objects, such as the processor, cannot be used simultaneously by both threads, and will appear in the state of either the parent or the child.

The parent thread might "loan" some of the objects it is using to the child for a period. For example, in the Berkeley Unix `vfork` operation, a parent process creates a child

process, and the parent process is then suspended, with the child using the parent's address space resources, until such time as the child executes a new program image [LEFF88]. At this time, the child is allocated its own address space, and the parent's address space is returned. The `vfork` operation can be represented by a fork vertex. Some objects present in the state of the incoming edge (which represents execution of the parent thread) are also present in the state of the outgoing edge that corresponds to the child thread, but are absent from the state of the outgoing edge that corresponds to the parent thread. At some later time, when the child is allocated resources of its own, the child sends a message to the parent thread which enables the parent thread to continue. The reception of the message is represented by a join vertex, with the state of the outgoing edge (which represent execution of the parent thread) containing those objects loaned to the child.

The second type of event that involves two threads is where one thread waits for the termination of another. This situation can be treated as the terminating thread sending a message to the waiting thread.

(5) Two message sub-tasks

Message join/split events involve two or more message sub-tasks (see Subsection 4.2.4). A message join/split vertex has up to two outgoing message edges. The state of each edge is the state in which the message it represents was created.

(6) State change summary

It is possible to determine the state changes caused by each event in an interaction network, although for some event types this is not trivial. Where one thread creates another, the rules for determining the state of the outgoing parent and child edges depend on the implementation of the thread creation operation. Also, for source events that are thread-related, and for message receive events represented by a simple vertex, the current state of a thread at the time of the event must be determined so that the state of the outgoing edge can be established. One way to do this is to maintain a record of the state of each thread, so that when a thread becomes associated with a new task the state of the thread is available and can be recorded.

6.1.4 Event Selection

Clearly, a very large number of event types could usefully be recorded in an interaction network. The overhead of recording, storing, and analysing interaction networks containing a very large number of events is high. In any implementation of a monitor for recording interaction networks, only certain probes will be provided, with probes that detect structural events always provided. For any set of recorded interaction networks, some subset of the probes provided would have been enabled. Structural events will

always be recorded, as without them interaction networks cannot be produced. Events necessary in the identification of critical paths will usually be recorded, with other object-use event types recorded at the performance analyst's discretion.

6.2 Analysis of a Single Interaction

Each interaction network is a piece of performance data, that describes in detail the way in which a task was performed by a distributed system. The performance information that can be derived from an interaction network depends on what event types were recorded. We now consider some of the analysis methods by which performance information can be derived from interaction networks that contain structural events, and begin_use and end_use events.

The way in which performance information can be aggregated at different levels of detail is discussed first, as information from most analysis methods can be presented at different levels of aggregation. Several ways are then discussed by which performance information can be derived from the analysis of interaction networks. Some types of performance information are available from more traditional monitors. This information includes some performance indices, basic statistics, and program profiles. Other analysis methods have been developed for use with interaction networks, such as browsing of interaction networks. The most important techniques for analysis are summarised at the end of the section.

6.2.1 Levels of Aggregation

Most types of performance information derived from interaction networks can be presented at one of three levels: the task level, the node level, and the thread level. At the task level, statistics are calculated for the task as a whole. At the node level, a separate set of statistics is calculated for each node (for thread-related activities) and for each pair of communicating nodes (for message-related activities). At the thread level, a separate set of statistics is calculated for each thread (for thread-related activities) and one for each pair of communicating threads (for message-related activities).

Consider the performance index "Processor time", an index that applies to threads but not to messages. At the task level, the total processor time used during the task over all processors is reported. At the node level, the total processor time used by the task on each node is reported. At the thread level, the total processor time used by each thread on behalf of the task is reported.

Further levels of aggregation are useful in some situations. For example, in process-oriented systems, cluster group and/or cluster levels could be added between the node and

thread levels as appropriate. Where procedure call and return events are recorded, the procedure level would be added below the thread level.

6.2.2 Indices

Among the many performance indices that can be calculated from an interaction network are:

(1) Response time, which was defined in Chapter 5 as the difference between the times of the first and last events in an interaction network.

(2) Total times spent using, or queuing for, system resources. Such indices can be easily computed if begin_use and end_use probes are installed for each resource and its associated queue. Times can be aggregated at various levels (including task, node, and thread), and may be for the whole task, or just the activities on the critical path.

(3) Normalized processor time. Anderson and Lazowska define a performance metric called *normalized processor time*, for use in the tuning of parallel programs [ANDE90]. Normalized processor time is calculated for each procedure, and depends on the procedure's execution time **and** the number of processors that were executing useful work (processor time spent executing a spin-lock, for example, is not useful work) during execution of the procedure. The number of processors executing useful work is of interest, because a procedure whose execution occurs when few other processors are executing useful work is a potential cause of a low level of parallelism.

Normalized processor time can be calculated from an interaction network if the following event types are recorded: procedure call and return, begin_use(processor), end_use(processor), begin_use(spin_lock), and end_use(spin_lock).

(4) Reaction time, which has been defined [JAIN91] as the time between input of the last character of a command, and the start of the command's execution. In an interaction network, reaction time is the time between the user input event and the first event of the command's execution, which is often the first begin_use(processor) event in the interaction network.

(5) Response ratio, which is the ratio of the actual response time of an interaction to the response time of the same interaction executed under optimal conditions (minimum response time), as described in Subsection 3.4.2.

Of these five indices, response ratio is both difficult to measure and important. The difficulty lies in accurately estimating an interaction's minimum response time. One way of computing this time is to run the interaction under optimal conditions, but measuring all interactions under optimal conditions is not feasible. An interaction's minimum response time must therefore be estimated from information recorded during its execution.

For single-threaded interactions where there is no file caching, a good estimate of the minimum response time is given by CPU time plus I/O device times for file I/O. In distributed and parallel systems, where interactions are multi-threaded and many caching techniques are used, there is no easy way to estimate minimum response times. One way by which interaction networks can be used to estimate the minimum response time is to reduce an interaction network to represent only the processing and messages likely to be required in the optimal circumstances. Then, the execution of the reduced interaction network by the target distributed system can be simulated and the simulated response time measured. Further work is required to develop methods to estimate minimum response times from interaction networks.

Interestingly, for a program activity graph, the length of the critical path is an estimate of the minimum execution time of a program, because the program activity graph recorded for a program is intended to be the same irrespective of the workload when the program was executed. To achieve this, edges that represent process execution are weighted with the processor time, and edges that represent messages are weighted with an estimate of the minimum network delay for messages of that length. It is not clear how accurate these estimates are for minimum response time calculated from program activity graphs. In particular, no allowance seems to have been made for processing that occurs when a program is executed under typical workload conditions, but which is not required when the program is run under optimal workload conditions. For example, under optimal conditions, a program can have much more physical memory and will therefore generate many fewer page faults than under conditions of typical workload.

6.2.3 Basic Statistics

From interaction networks it is easy to derive simple statistics on event counts and durations of activities. Simple statistics related to communication are useful because the performance of a distributed interaction is heavily influenced by the communication that occurs. DPM provides statistics like message counts, size, and frequency (time density), both in total and for pairs of processes [MILL88a]. As message send and receive events are always recorded in an interaction network, it is only necessary to ensure that message size is recorded for events of these types to be able to calculate from interaction networks all of the communication statistics provided by DPM.

Another class of basic statistics are those on resource use times, queue residency times, and queue lengths. If begin_use and end_use events are recorded for the resources and queues of interest, then these statistics are easily calculated from an interaction network.

6.2.4 Response Time Decomposition

To determine the reasons for poor interactive performance, a valuable approach is to try to answer the question "Where did the response time go?", by finding the components of response time [PENN86], [PENN88], [ANDR89]. To reduce response time, the activities that contribute most to response time must be identified, and their durations reduced. In response time decomposition, the response time of an interaction is divided up into the times spent in each of a disjoint set of states, as described in Subsection 3.4.3. At a finer level of detail, there might be information on the individual state transitions. The problem then becomes one of defining the states, determining the time spent in each state, and presenting the decomposition in a useful form.

In earlier work to which the present author contributed [PENN86], [PENN88], response times were decomposed into resource use and resource queuing times, with state definition corresponding to the system-level viewpoint discussed in Subsection 6.1.2 of this thesis. The components reported were the times spent in each state. A monitoring system known as ASHMON was constructed by the author to record the response time decomposition for sets of interactions [ASHT84].

ASHMON was developed for a centralised system, where each interaction was performed entirely by a single process. States were analysed at the system level, with each state representing a process either using a resource or queuing for one. The number of states was small: running on the processor, queuing on one of four processor queues, running a disc request using one of two disc controllers, queuing for a disc request on one of two disc controllers, or queuing on one of five different categories of system queues. The states for disc controller running and queuing were further decomposed into the times spent running on, and queuing for, requests for each disc and each disc partition.

Response time decomposition reported by ASHMON listed the percentage of response time spent in each state. This report was useful because of the relatively small number of states: 24 in total when the disc states were reported at the most detailed (disc partition) level. Deriving response time decompositions for a distributed system, as the interaction network is designed to facilitate, is much more difficult, as:

(1) Tasks are usually multi-threaded. At any point in a task, several sub-tasks can be proceeding in parallel. The state of a task at any instant is given by the union of the states of all of its sub-tasks at that time. For example, consider a time where three sub-tasks exist. Each sub-task can be in one of three states: running, ready, or blocked. The task at this point can be in one of up to 27 different states (<running, running, running>, <running, running, ready>, ...). In practice, some of these states may not be possible. For example, if all sub-tasks are executing in a uniprocessor node than any task state that

contains more than one sub-task in running state is infeasible. Breaking down response time into twenty-seven states is unlikely to provide a very clear picture of "where the time went". In a single-threaded environment, however, with the same three sub-task states, decomposing response time into these three states would give a good idea of where the time went.

(2) The number of possible states is much larger. The state of a sub-task in an interaction network has been defined as the set of objects currently in use by that sub-task. This set is potentially very large, including all system resources and resource queues, all procedures in all programs, all files, and all database objects. Also, it is possible for a state to show that many objects were in use simultaneously, which was not the case with the system-level objects monitored by ASHMON, where each state corresponded to the use of a single object. For n objects, there are n states if 1 object can be in use at any one time, but 2^n states if any combination of objects can be in use at any one time. The large number of objects and the fact that many objects may be in use simultaneously, together result in a huge number of possible states of a sub-task.

A simple decomposition of response time into the various states observed during a task is therefore unlikely to be of much use to a performance analyst. Two techniques that will give useful response time decompositions from interaction networks are now discussed. In the first technique, states are defined based on sub-task states, with the state times reported being the aggregation of times over zero or more sub-task activities. In the second technique, response time decomposition is based on task states. Both techniques can be applied to all activities represented in an interaction network or to some arbitrary subset of those activities.

(1) Decomposition based on sub-task state

We now describe a four step procedure, based on sub-task states, that can be used to provide a very general response time decomposition. Briefly, the steps are: (i) to select what activities to analyse; (ii) to select the objects that define what we will call the simple states; (iii) to specify compound states, states of interest to a performance analyst in a particular evaluation, based on these simple states; and (iv) to specify how information on times spent in these compound states is to be reported. These four steps are now explained:

(i) Selection of activities. The activities to analyse can be selected using a boolean expression, the activities being those whose state satisfies that expression. Some examples are given in Table 6-1. To allow critical path analysis, activities on the critical path can be regarded as having the "CP" object present in their state.

Boolean expression	Activities selected
true	All activities
Node N_A **or** Node N_B	All activities performed on Node N_A or Node N_B
count(file objects) ≥ 10	All activities during which ten or more file objects were in use
CP	All activities on a critical path
CP **and** (processor or processor queue)	All activities on a critical path during which a processor or a processor queue was in use

Table 6-1: Possible boolean expressions for selection of activities

(ii) Definition of simple states. In this step, a set of *objects of interest* is selected that is to define the *simple states*. Each simple state is defined by the objects of interest in use, and records use of only those objects relevant to the current analysis. Where the set of objects of interest is relatively small, the number of possible simple states is much smaller than the number of possible activity states.

Consider an example. For one analysis, a performance analyst selects all of the procedure objects and one type of system resource object, the processor objects for instance. A decomposition based on these states will report, for each procedure, the times spent using each processor, and the time spent not using any processor.

(iii) Definition of compound states. A *compound state* is defined by a *state expression*, a boolean expression similar to those described in (i), but referring only to objects already defined to be of interest. The time spent in each compound state is equal to the sum of the times spent in each simple state that *matches* the state expression that defines the compound state. For example, suppose that * selects all objects of the specified type. The time spent in the compound state defined by the expression "processor(*) and procedure(blarg)" is equal to the amount of processor time spent by all processors in procedure blarg during the activities selected in (i).

In some analyses, it may be acceptable for compound states to overlap. That is, the time taken by some activities may be counted in more than one compound state, while the time taken for other activities is not counted at all. Often, however, it is required that each activity selected in (i) be included in exactly one compound state, that is for compound states to be disjoint. If compound states are not disjoint then, for example, the total time spent in all compound states for activities on a critical path will not equal response time. A useful feature of an analysis tool is, therefore, the ability to check that all the specified compound states are disjoint.

(iv) The performance analyst can now be presented with the times spent in each of the compound states specified in (iii), summed over all activities selected in (i) .

A performance analyst may use this approach directly to determine response time components. A tool to do this would provide a number of facilities to make decompositions easy to specify. For example, a number of object groupings could be pre-defined for use in steps (i) to (iii), with the analyst able to define additional groups. Then, if the analyst wanted to select all activities in which a processor was used, the processor group could be specified rather than "processor 1 or processor 2 or .....". If all procedure objects are to be used in definition of simple states then the analyst could select the procedure group rather than having to select all procedure objects individually. Also, libraries of useful state expressions can be built up and used in many different analyses.

More often, perhaps, an analyst would use higher level tools in response time decomposition. Based on some higher level description of the analysis to be performed, these tools would specify on the analyst's behalf: the activities to analyse, the simple states, and the compound states. For example, a performance analyst might request a decomposition of the time spent on the critical path into times spent using each combination of hardware resources and queues. Or the analyst might request a histogram of the time spent with 1, 2, 3, ... files open. A higher level analysis tool would then perform steps (i) to (iv) to satisfy the request.

Additional information about the way in which an object was used might be recorded in an interaction network. For example: the target cylinder might be recorded for every disc access; and the message length might be recorded when a message object is created. This information could be made available during analysis as properties of an object. For example, in selection of activities an analyst might specify the expression "message.length $\geq$ 100". This expression would match all activities that occurred for a message that was at least 100 bytes long. The expression "disc(cylinder) = 0" would match all activities where a disc request to cylinder 0 was in progress.

Where all activities on a sequential path through an interaction network are analysed, and compound states are disjoint, the state times reported are the components of the path's duration, that is their sum is equal to the length of the path. Any activities that occur in parallel with the selected path are ignored. Critical path(s) and near-critical paths are very important sequential paths. The critical path is a logical choice for use in the decomposition of response time, as to reduce the response time of an interaction some of the activities on each critical path must be reduced in length.

Where the interaction network (or component) being analysed contains parallel activities, the decomposition reflects an overall view of the activities of a task. The sum of

the components may be greater than response time as the durations of activities that occur in parallel are included in the times reported. This type of analysis can provide information on all of the activities that are performed within a task. For example, the total hardware resource usage and queuing times across all activities can be determined.

(2) Decomposition based on task state

In (1) above, sub-task states were used for response time decomposition, and the overall task state was not considered. Another viewpoint for analysis is to consider that a task has a single state at any point in its execution, with that state composed of the states of all of its sub-tasks in existence at that time. No double-counting of state times occurs where task states are analysed, as a task has a single state at any point in time.

A task state is considerably more complex than a sub-task state, as a task state is made up of a varying number of sub-task states. As pointed out earlier, treating a task's state in this way can result in a potentially very large number of states.

The approach outlined in (1) can be used for task state analysis, but some enhancements are needed to support the multiple sub-task nature of the task state.

First, let us consider when changes in the state of a task occur. A task changes state when the state of one of its sub-tasks changes, or when the number of sub-tasks in the task changes. Between two changes in state, the state of a task consists of the union of sub-task states for all of the sub-tasks in the task.

To match state expressions, the set of objects that may be used by any task is simply the union of the sets of objects that may be used by any of its sub-tasks. To detect periods of low parallelism in a parallel program, an analyst might define a compound state by the expression count(processors) <= 1 to determine the amount of time during which 0 or 1 processors were in use. Or the analyst might ask for a histogram showing the total time for which 0, 1, 2, 3, ... processors were in use. To determine the reasons for low parallelism, the analyst could focus on just those periods during which 0 or 1 processors were in use.

We now define two functions useful in state expressions for analysis of task states: stnum() and stcount(). The stnum() function returns the number of sub-task states that go to make up the task state. The state expression stnum() > 4 matches all task states where more than 4 sub-tasks exist. The stcount() function takes a state expression *sp* as a parameter and returns the number of sub-task component states that individually match *sp*. For example, if any task states are matched by the expression "stcount(CP) != 1" then there is a bug in the algorithm that determines the critical path!

(3) Summary

The approach outlined above for response time decomposition is very general, and allows a wide range of decompositions of both sub-task and task states. Decompositions of the sub-task states of the critical path are important as they highlight the types of activities that contribute most to response times. Decompositions may be performed for all of an interaction network, or some arbitrary part of it. Decompositions may be reported as the total time spent in each compound state, or the time in each compound state may be reported at the node or the thread level.

Some types of decomposition will be used often enough to build into an analysis tool. Examples include: decomposition into system-level state times, as in the earlier work described in [PENN86], [PENN88]; decomposition into times spent in different procedures; and, for "objects" in an object-oriented system, decomposition into times spent in each object type, object instance, and object method.

To allow response time decompositions to be performed, an interaction network must include begin_use and end_use events for all objects that are to be used in decompositions.

6.2.5 Profiling

Program profiling is a well established technique [GRAH82], which directs attention at those parts of a program which consume most processor time (see Subsection 3.4.3). Some profilers report the amount of processor time used in the execution of each procedure, or perhaps the time spent in various address ranges in a program. Often, the profile is built up by the clock interrupt handler sampling the program counter of the currently running process. Other types of program profilers may report counts of statement executions or invocations of procedures.

A simple type of profile is that where processor time is divided according to the amount of time spent executing each procedure. This type of profile can be derived from an interaction network as a form of response time decomposition, if procedure begin_use (call) and end_use (return) events are recorded. A shortcoming of this type of profile is that, as much time is usually spent executing library functions called from many places, knowing that most time was spent in library functions does not provide much information about the most processor-intensive areas of a program.

A way of addressing this problem is to report for each procedure the sum of the time spent executing the procedure and any time spent on its behalf through calls to other routines. A widely used profiler of this type is gprof [GRAH82], which computes the time spent executing in each procedure using the sampling technique outlined above. The time spent executing on behalf of procedures is calculated by recording the call graph, which

specifies who called whom and how many times. The value of this technique is greatly lessened because it requires an assumption that all calls to the same procedure take the same amount of time.

It is possible to derive "time spent on behalf of" profiles from interaction networks because:

(1) The call graphs for each thread used can be extracted from one or more interaction networks.

(2) The processor time for each procedure is easily determined from begin_use and end_use events for procedures and processors.

Profiles with respect to many other objects can also be derived easily from interaction networks. One of many possible examples is the time spent performing disc requests by and on behalf of each procedure.

6.2.6 Prediction

Interaction networks can be used to predict the impact of tuning actions, if the effects of those actions can be modelled as shortening the times for one or more activities, and/or the elimination of one or more activities. For example, the act of doubling the speed of the CPU can be modelled by halving the length of all CPU run activities. The act of increasing memory size can be modelled by removing a proportion of local and remote disc accesses.

Once the impact of some tuning action has been specified in these terms, the interaction network has to be adjusted to reflect these changes. The critical paths of the original network and the adjusted network can then be compared.

6.2.7 Animation

Tools for displaying animations of various aspects of a program's execution can be valuable, and an interaction network contains enough information to support many different types of animation. Joyce *et al* describe Mona, a tool for displaying an animated graphical view of the interprocess communication in an application system [JOYC87]. Mona represents each process with an icon, and each message between two processes with an arrow from the sending process' icon to the receiving process' icon. An interaction network does contain enough information for this sort of animation.

A more general framework for animation of information from interaction networks can be based on the mechanisms for response time decomposition, discussed above in 6.2.5. For each compound state selected for an animation, two animation actions can be defined: one to be performed on transitions into the compound state, and the other to be performed on transitions out of the compound state. Each animation action might involve activities

like adding or removing a screen object, moving a screen object, and changing the display attributes of a screen object. Where sound output is available, audio output is another possible type of animation action.

6.2.8 Browsing

An *interaction network browser* is an analysis tool that presents a graphical representation of an interaction network to a performance analyst. A browser displays a digraph representation of an interaction network, and provides facilities for a user to access performance information derived from it.

A browser must be able to display all events and activities of an interaction network. A useful display format was described in Subsection 4.2.5. A browser should also provide facilities to zoom in and out, and to pan around the interaction network display. The critical path through the interaction network can be highlighted by using a different brush for edges on the critical path.

A full display of an interaction network, with all vertices and edges displayed, provides a complete picture of a task's execution. Because large interaction networks can contain a very large number of vertices, a browser must be able to manage the complexity of large networks. Several methods for doing this are listed below.

(1) An interaction network can be displayed at one of three levels of abstraction: node, thread, and full. At full level, all vertices and edges are displayed. At thread level, threads and message passing between threads are highlighted, so all simple events internal to a thread or message are suppressed. In a thread-level view of the interaction network in Figure 4-6, Vertex v_{11} would not appear. Usually, a much greater proportion of vertices would not appear in the thread-level view. The interaction network in Figure 4-6 contains virtually no begin_use or end_use events.

At the node level, only periods during which a sub-task was present on each node and interactions between nodes need be highlighted. Figure 6-2 shows a node level view of the interaction network resulting from the combination of Figure 4-6 and Figure 4-8. Note that the scale of the time dimension has not been preserved, and the vertices have been renumbered. Varying the thickness of node-related edges could be used to indicate the number of sub-tasks executing on each node at any particular time.

(2) The number of vertices to be displayed could be reduced by showing only those vertices needed to determine transitions into and out of compound states.

(3) Simple filters could be provided to suppress events recorded for selected nodes or threads, or of selected types, or relating to use of objects in selected groups.

THE LIBRARY
UNIVERSITY OF CANTERBURY
CHRISTCHURCH, N.Z.

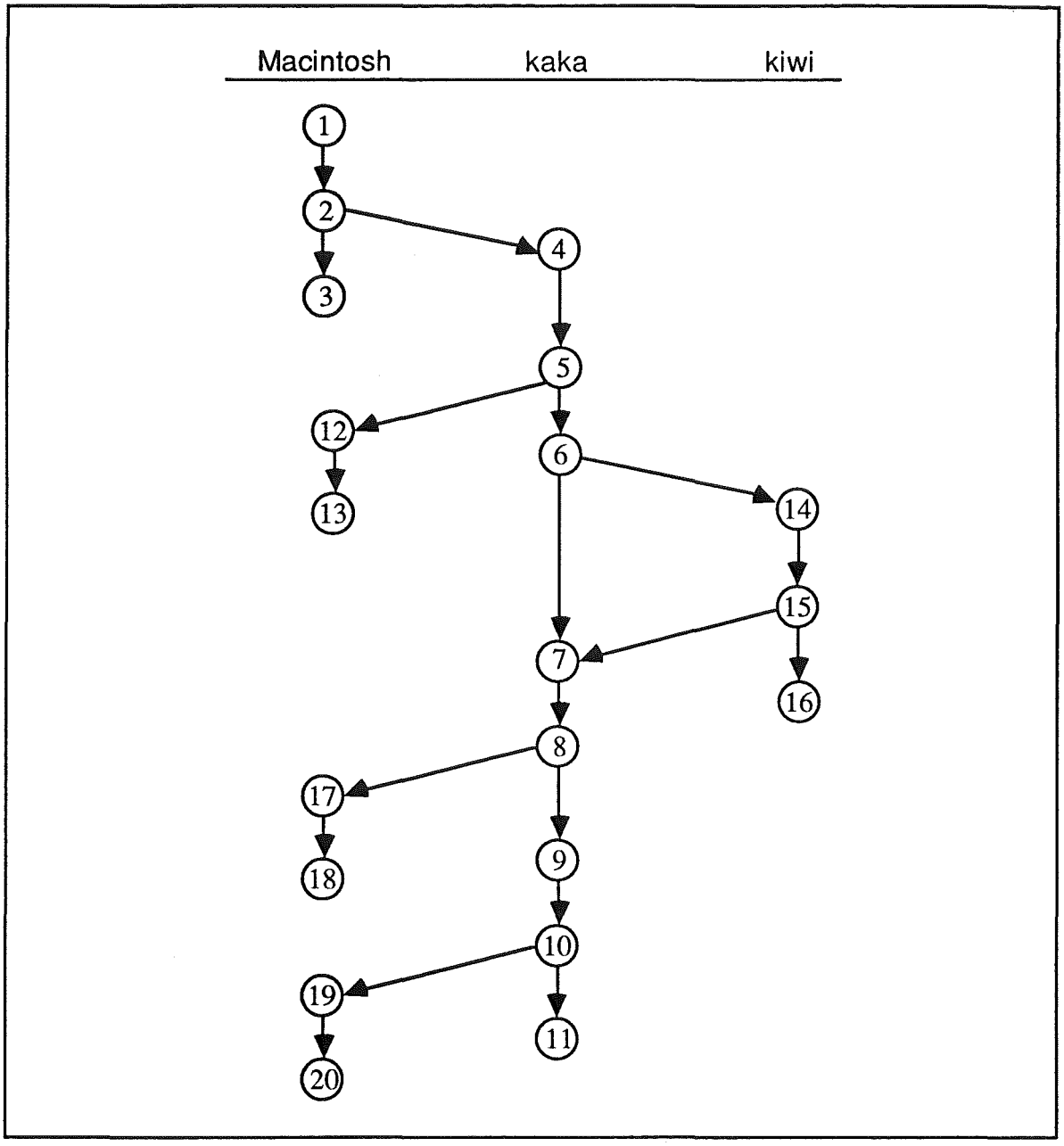


Figure 6-2: A node level view of an interaction network

An interaction network browser could also provide a single interface to all of the other types of analysis tools discussed in Section 6.2. For example, an interface for specifying response time decompositions could be provided, with compound states summarised in a table or indicated on the interaction network display in some way, perhaps by using different colours for different states. In such a system, the analyst might be able to select the activities to be analysed using a mouse. Also, in a what-if analysis, activities to be eliminated, or whose length is to be reduced, could be selected using a mouse. Finally,

alternative final vertices (see Subsection 5.3.4) could be selected using a mouse, and the browser could then show the critical path from v_0 to the selected vertex.

An interaction network browser was developed by Thoo Lip Chau [THOO91], to display interaction networks that the present author had recorded.

6.2.9 Use in Trace-driven Simulations

Because an interaction network will contain a very detailed record of the events that occurred during an interaction, event records extracted from one or more interaction networks could be used as input to trace-driven simulation programs. Interaction networks would be a particularly useful source of input where a trace-driven program simulates some aspect of the execution of individual interactions.

6.2.10 Summary

Many ways in which performance information can be extracted from an interaction network have been discussed. One very important technique is that of response time decomposition. A very flexible method for specifying response time decompositions has been outlined. An interaction network browser would also be a very powerful tool for performance analysis.

6.3 Analysis of Sets of Interactions

Sets of interactions can be analysed using techniques that are much the same as those described in Section 6.2. An analyst must be able to specify the set of interactions to be analysed, the set being all or some of the interactions that occur in a distributed system within some time interval. Analysing all interactions in some interval provides performance information for the whole distributed system. Analysing a sub-set of interactions provides performance information on, for example, selected classes of: interactions, nodes, programs, network links, hardware resources, and queues.

Some method must be available for specifying the set of interaction networks to be analysed. Selection can be based on the specification of a state expression, with all interaction networks that contain at least one activity that matches the state expression included in the set. For example, the state expression "true" selects all interaction networks. The state expression "Node N" selects all interaction networks that contain at least one activity that was executed on Node N. The state expression "Program P" selects all interaction networks that contain at least one activity that was executed in Program P.

Once a set of interactions has been selected, performance information can be derived about this set using all of the techniques described in Section 6.2, except that of browsing.

For these techniques, the values reported are the result of aggregating the values computed for each interaction network in the set.

Analysing sets of interactions allows complete performance data to be gathered for objects used across many different interactions. Overall resource utilisations and queue lengths can be calculated. All traffic on a network link can be identified and, say, subjected to protocol analysis. Overall performance information can be calculated for programs (and threads) that participate in many interactions, in particular server programs and highly interactive programs.

A very important type of performance information that can be determined from analysis of sets of information is critical path components aggregated over all interactions in the set. This type of performance information is very important in system tuning, as large components highlight potential system bottlenecks. By aggregating the critical path components from (potentially) very many interaction networks, we have minimised the effects of cases that exhibit some of the problems in critical path analysis described in Subsection 5.3.3.

We consider that one of the most important uses of interaction networks will be in determining the critical path components for large sets of interactions. In conclusion, a very large amount of performance information can be derived from performance analysis of both individual interaction networks and of sets of interaction networks.

6.4 Existing Monitors for Distributed Systems

We now consider the performance information produced for distributed and parallel systems by monitors described in the literature. We then compare the range of information provided by these monitors with the types of performance information available from analysis of interaction networks.

6.4.1 Performance Monitors

Performance information available from IPS-2, DPM, TMP, JADE, and Monit is discussed in this subsection.

(1) IPS-2

IPS-2 is an "interactive, trace driven performance measurement system for parallel and distributed programs" [HOLL91a], [HOLL91b], [MILL87], [MILL88a], [YANG88]. In IPS-2, a program is modelled as a five level hierarchy. A *program* is executed on one or more *machines*, on each of which executes one or more of the program's *processes*. Each process executes *procedures* and, during the execution of each procedure, *events* may be

recorded. Analysis can be performed at the program, machine, process, and procedure levels of aggregation, with this hierarchy of analysis levels known as the *measurement hierarchy*.

IPS-2 can calculate approximately 25 metrics, including the number of processes and machines, response time, speedup, message and file I/O rates, CPU time, and counts and times spent in various queues. For a component at a selected level of the hierarchy the overall value of the metric can be reported, or a plot of the value of the metric against time can be displayed.

IPS-2 can also construct a Program Activity Graph to represent the execution of a program, and can find the critical path through it. Decompositions of the time spent along the critical path, and of the total program time, are available at any of the four levels. Components include CPU time, synchronisation waits, and message delays. A what-if mechanism is available whereby a user can select one or more activities on the critical path. These activities are then given a weight of 0 and the critical path is recalculated. IPS-2 does not provide for displays of program activity graphs, as its authors consider that the number of nodes involved (estimated at tens of thousands) would make this impractical.

Phases of the execution of a program can be identified, and performance metrics recalculated for selected phases. A facility for automatic phase identification is provided, a phase being defined as an interval where the levels of one or more metrics contain similar values throughout that interval.

(2) DPM

DPM (Distributed Programs Monitor) "is a system for monitoring the execution and performance of distributed programs", [MILL86], [MILL88a]. DPM was developed by Barton Miller prior to his work on IPS-2. A distributed program consists of one or more processes which communicate by message passing, and which are spread over one or more machines. DPM does not use a measurement hierarchy similar to the that of IPS-2. The events recorded by DPM occur in process management (create, start and stop of execution, and terminate), and in interprocess communication.

DPM can calculate a number of basic communication statistics, that provide information on: the pairs of processes that communicate, and the size and frequency of the messages between each pair; the length of the message queues associated with each process; and the amount of time messages spend in a message queue. Techniques that display communication activity graphically in real-time have been developed. Processes and communication links are represented either in a graph, or as a matrix. The colour of each link and process changes from violet to red as the amount of work performed by the link or process increases.

An interesting feature of DPM is its ability to track the *path of causality* for a request to a server (see Section 4.3). A path of causality shows the message passing that occurred between the processes of a multi-process server in the processing of a single request to the server. Detecting these paths requires the construction of *program history graphs*, which represent execution and message passing using a digraph. Program history graphs have been used in the analysis of a file server whose functions were split across 4 processes. Based on program history graphs for a large number of server requests, a diagram was constructed to show the message passing behaviour expected of the four processes in response to a request.

DPM can also be used to measure parallelism (speedup), calculated as total CPU time / response time, where response time is the length of the longest path (the critical path) through a program history graph.

(3) TMP

TMP is a hybrid monitor constructed for the INCAS distributed system project [WYBR88], [HABA90]. TMP monitors distributed programs, with each program consisting of a (programmer-defined) hierarchy of sub-systems. Each sub-system consists of process clusters and/or other sub-systems, with each cluster containing one or more processes. Each machine executes one or more process clusters.

Each node in the system incorporates a Test and Measurement Processor (TMP), that records events and produces local summaries from the recorded events. Each TMP reports, at regular (synchronised) intervals, its local summaries to a central analysis node over a network used only for performance monitoring purposes. A TMP summary consists of metrics calculated over the period since the previous summary was reported. Process and cluster metrics include processor time, times spent in various queues, system time, and message and file access statistics. Machine metrics include processor and idle times, queue times and lengths, and message statistics.

All program and sub-system metrics are computed by the central analysis station. These metrics include total elapsed and processor times, number of machines, and parallelism. The central station provides a graphical interface that allows the user to move around the program hierarchy, and to display various metrics at each level. In one type of display four sub-systems and their interconnections are displayed. Each sub-system is represented by a box enclosing a bar chart of four sub-system metrics. Each interconnection is represented by an arc between two sub-system boxes, with the amount of traffic on the link indicated by the thickness of the arc. This display can be animated.

A second display contains a summary of machine information, with a bar chart and two Kiviat graphs [FERR83] displaying machine-level metrics for three machines.

(4) JADE

JADE is a distributed programming environment that includes monitoring tools [JOYC87]. The JADE monitoring tools record information about distributed programs that are linked with the instrumented version of the JADE communications library. In JADE, a distributed program consists of a set of processes that communicate using message passing. Several different *consoles* have been written to provide performance information from the event records collected.

The *text* console prints a few lines of text describing each event that occurs. Filters can be defined to limit the number of events described. Breakpoints can be defined, and a sequential debugger can be used on any process stopped at a breakpoint.

The *Mona* console provides an animation of message passing between the processes of an application. Processes are represented by icons, and messages by directed arcs between two process icons. Icons can be moved, and can be grouped. Message passing within a group is not displayed.

The *event line* console displays an event line for each process. A process' event line shows a history of recent events, with each event represented by a code of up to three characters. Periods between consecutive events in a process are represented by dashes if the process is executing, and by dots if the process is blocked on a communications library call.

Other analysis tools include a console that computes statistics on message passing, a deadlock detection console, a protocol checking console, and a method for deriving composite events from a stream of primitive events.

(5) Monit

Monit analyses trace files recorded during the execution of a parallel program, and produces graphs of resource and system queue usage against time [KERO87]. Monit's main output is a display consisting of a number of bar charts, presented side by side, with time increasing down the vertical axis. Variables that can be graphed include: the utilisation of, and queue lengths for, processors, discs, and memories; queue lengths for other queues; and the values of program variables. In addition to producing histograms, Monit can compute several statistics for each variable.

6.4.2 Comparison

Nearly all of the types of performance information described in Subsection 6.4.1 can be determined from analysis of interaction networks. These include:

(1) Indices such as response time, resource use and queuing times, and indices relating to interprocess communication.

(2) Event traces.

(3) Various types of response time decomposition.

(4) Profiling.

(5) Replay.

(6) Animations.

The major differences between methods based on interaction networks and the information produced by the other monitors described are that:

(1) Interaction networks provide performance information about interactions. None of the tools and techniques discussed in Subsection 6.4.1 are designed for performance analysis of interactions; all are designed for analysis of programs. Also, through analysis of sets of interaction networks, performance information can be derived for an arbitrarily selected sub-set of the workload of a distributed system. Examples of such sub-sets are: all interactions that made requests to a file server, or all interactions that included execution on a particular node.

(2) A very general method has been described for deriving response time decompositions from interaction networks. Decompositions are based on use of objects, where an object is any system resource whose use can contribute to response time. Response time decompositions available from other monitors are restricted to decompositions of a few pre-determined types, such as decomposition with respect to use of hardware resources, or decomposition with respect to time spent in each procedure of a program.

(3) Although critical path analysis can be used in analysis of both interaction networks and program activity graphs, the performance information derived from each is very different, as described in detail in Section 5.4.

(4) Although many examples have been given of acyclic digraphs being used to represent computation and communication in a distributed environment (see for example [MILL88a], [MILL89b], [MILL90a], [TSAI90]), we know of nothing similar to the interaction network browser developed as described in Subsection 6.2.8. Tsai *et al*

comment that a graphic representation of an IPEL can provide the user with a "better understanding of a program's run-time behaviour" [TSAI90], although it is not clear whether a program for displaying IPELs has been developed (IPELs are described in Section 4.3). Early experience with a prototype implementation of an interaction network browser, discussed later, indicates that a display of an interaction network provides useful performance information.

6.5 Summary

At the start of this chapter, we defined the state of a sub-task activity to be the set of objects in use during that activity. This object-use framework provides a very general way of defining states of sub-tasks and tasks. To support recording of states, begin_use and end_use event types were introduced.

Many methods have been described for the performance analysis of individual interaction networks. Two of the most important are a very general method for producing response time decompositions, and the interaction network browser. Most of these analysis methods can also be applied to sets of interactions that form some arbitrarily chosen part of the workload of a distributed system during some period. The response time decomposition of the critical paths of a set of interaction networks can be used, for example, in bottleneck detection.

The most important difference between the author's method and other methods of performance analysis for distributed systems, is that interaction networks provide information on interactions, with most other methods restricted to program analysis only. The very general method for response time decomposition, and the interaction network browser, seem to be unique.

This chapter completes the conceptual framework for our approach to the problem of measurement of interactive performance in distributed systems. Issues to do with the implementation of monitors to record interaction networks are discussed in Chapter 7. Subsequent chapters will describe INMON, a prototype monitor constructed by the author to record and analyse interaction networks. Finally, some performance studies done using INMON will be presented.

Chapter 7

Implementation

So far, little has been said about how interaction networks might be recorded. We now discuss possible approaches to that problem.

An interaction network consists of a set of event records, and of links between event records. A global timebase is needed to ensure that timestamps in event records from different nodes are based on a consistent set of clocks. This separate, but fundamental issue, is introduced in Section 7.1. Approaches to the design of an *interaction network monitor*, a monitor that can record and analyse interaction networks, are discussed in Sections 7.2 to 7.4. Ways of structuring *distributed monitors* (monitors developed for use in distributed systems), and ways in which an interaction network monitor could be structured, are described first in Section 7.2. Likely components of an interaction network monitor are then described, with design of the probes that detect events discussed in Section 7.3, and the event recording and performance analysis functions covered in Section 7.4.

A demonstration is required that shows that a monitor can be implemented to record interaction networks. The author has designed and constructed INMON, a prototype monitor for recording interaction networks. INMON will be briefly described in Section 7.5, and examined in detail in Chapters 8 to 11. Clock synchronisation is discussed in Chapter 8. The probes and the event recorder are covered in Chapter 9, with discussion of analysis tools in Chapter 10. In Chapter 11, results are given for experiments performed using INMON.

7.1 Clock Synchronisation

Methods developed in Chapter 6 for analysis of interaction networks are based on the assumption that events are timestamped by a global clock. If a global clock cannot be used to timestamp all events, then the following problems arise:

(1) It is difficult to determine the duration of inter-node message transmissions.

(2) If there are sink vertices representing events that occurred on different nodes then it is not possible to accurately determine the v_f vertices.

In practice, a global timebase cannot easily be provided in a loosely-coupled distributed system. The local clock in each node is typically crystal-controlled. True physical time can be taken as the international time reference TAI (Temps Atomic International) provided by the Bureau Internationale de l'Heure [ASTR84]. The time recorded by crystal-controlled clocks diverges from true physical time at a rate which is of the order of 10^{-6} seconds/second [CARL88], [KOPE87]. If no corrective action is taken, the difference between the clocks at various nodes will steadily increase.

The problem of providing a global timebase in a distributed system, the *clock synchronisation* problem, has been heavily researched. Many solutions have been proposed based on additional hardware and/or additional software. The maximum error (that is, the maximum difference between the clocks of any pair of nodes) predicted for various solutions varies quite widely. In the recording of interaction networks, solving problems (1) and (2) described above requires clocks to be synchronised to such a degree that the maximum synchronisation error is small compared to network transmission times. With current LAN transmission times in the order of milliseconds, the maximum clock synchronisation error should be of the order of one to two hundred microseconds.

The clock synchronisation problem is not central to this work, as an interaction network monitor can use any high-resolution clock synchronisation method. However, clock synchronisation with sufficiently small maximum error was not readily available for INMON, so the author has developed a new solution. Because the clock synchronisation problem is not central to this work, further discussion is largely isolated in a single chapter, Chapter 8, which contains detailed background on the clock synchronisation problem, and a description of our solution.

7.2 Monitor Structure

This section describes common structures for distributed monitors, in terms of monitor components and relationships between components. A general framework for the structure of distributed monitors is given in Subsection 7.2.1 followed, in Subsection 7.2.2, by discussion of the structure of some monitors described in the literature. Possible structures for an interaction network monitor are given in Subsection 7.2.3.

7.2.1 Structure Overview

Event-driven and sampling monitors were described in Subsection 3.2.1. An event-driven monitor must be used to record interaction networks, as an interaction network contains a set of event records. This subsection, therefore, discusses only event-driven monitors. Information on events is collected by a set of event probes. Each *probe* is a

fragment of code added to a system or user program. When a probe executes it records information about the event that has occurred. A set of probes can produce a stream of *event records* at each node in a distributed system.

The information recorded by probes is ultimately destined for analysis tools, and any performance database(s) maintained by the analysis tools. These analysis tools provide performance information derived, directly or indirectly, from the data in one or more event records. The tools gather information from one or more nodes in a distributed system. A performance analyst interacts with the analysis tools through a user interface. The interface allows the analyst to specify the performance data to be collected, to request displays of performance information derived from collected performance data, and to save performance data in the performance database.

Between the probes and the analysis tools may be one or more intermediate layers of software, responsible primarily for relaying information from the probes to the analysis tools in an efficient manner. In addition to relaying information, an intermediate layer might also perform some analysis.

Without intermediate components, probes provide event records directly to analysis tools. Because of the very high event rates observed in distributed systems, such configurations are seldom feasible. High event rates are handled better by introducing intermediate layers capable of event record filtering and/or more complex analysis, to reduce the volume of data that must be relayed to the analysis tools.

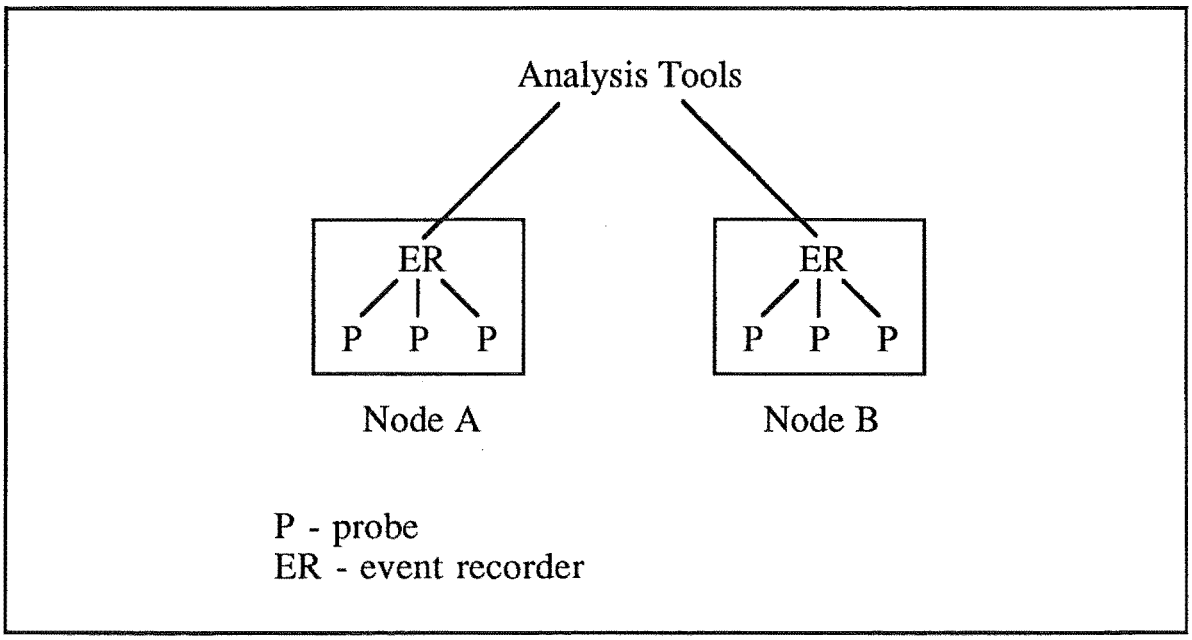


Figure 7-1: The structure of a hypothetical distributed monitor

Often, configurations of monitor components exhibit a tree structure, with the analysis tools and database at the root, and the probes at the leaves. Control flows downward from the analysis tools, and data flows upward from the probes to the analysis tools. Figure 7-1 shows the structure of a hypothetical distributed monitoring system. The programs executed on each node in a distributed system contain a set of probes. Each probe reports the fact that an event has occurred to the node's event recorder. An event recorder filters out any events that are not currently of interest, and stores the remaining event records in buffers. When a buffer is full its contents are transmitted to the analysis tools.

Communication between monitor components may be on-line, usually through procedure call or message passing interfaces, or off-line, perhaps through intermediate files. In the example shown in Figure 7-1, communication between a probe and an event recorder is on-line. Communication between an event recorder and the analysis tools might be either on-line or off-line. If communication is on-line, the analysis tools are responsible for display and/or archiving of the information in the event record stream. If communication is off-line, each event recorder writes selected event records to a file. Later, during analysis, the analysis tools must be able to identify sets of "related files". The files recorded on a group of nodes during some measurement session are a set of related files, as between them they contain all event records that were recorded during the session.

7.2.2 Structures of Existing Monitors

The structures of the distributed monitors IPS, DPM, TMP, and JADE (all introduced in Subsection 6.4.1) are now discussed to show that the hypothetical monitor structure described in Subsection 7.3.1 is representative of current practice.

(1) IPS

Descriptions of the structure of IPS are to be found in [MILL87] and [YANG89]. The event probes in IPS are called *agents*. Each agent stores event records in an area of the memory called the *data pool*. Each node has a *slave analyst* process that is responsible for managing access to the data pool on that node. A slave analyst computes several metrics describing performance on the local node, based on the information in its data pool. There is a single *master analyst*, that communicates with the slave analysts, and with the interface program through which the user communicates with IPS. The master analyst maintains several metrics describing overall program performance, derived from information from the slave analysts.

In IPS, slave analysts are responsible for performing much of the analysis. For example, it is not the case that the events that make up a program activity graph are all

transferred to the master analyst. Instead, analyses of program activity graphs, such as critical path analysis, are performed in a distributed fashion by the slave analysts. Event records are sent from a slave analyst to the master analyst only if the user requests some detailed information on some part of a program's execution. Keeping the transfer of event records to a minimum is seen as an important way of ensuring that the overhead of IPS is reasonable. [MILL87] and [YANG89] do not say whether PAGs may be saved in a performance database for later analysis.

(2) DPM

In DPM [MILL88a], a number of *meter probes* were added to an operating system to record events of interest. Each probe is a procedure call to a software module (*meter module*) that is responsible for passing event records to a *filter* process. The meter module buffers event records so that it can send them to the filter process in batches, thereby reducing the number of messages to be sent. Typically there is one filter process per distributed program. The filter selects event records to store in a log file, with selection based on a general, one-level, pattern-matching algorithm. Tools can be used to analyse the information in a log file at any time after the distributed computation has finished.

Other configurations are possible. For example, when network load is heavy there can be a filter process on each node, with the trace files merged when the computation being monitored has terminated.

(3) TMP

In TMP [HABA90], a hybrid monitoring approach is used. A hardware subsystem dedicated to monitoring, a Test and Measurement Processor (TMP), is incorporated into each node. The main components of a TMP include an interface to the system bus, an Event Processing Unit (EPU), a CPU, and a network interface. Probes are installed in the target software of a node. Each probe is one or more assembly language statements of the form STORE ADDR, VALUE. For each processor in a node, a range of 256 addresses is reserved for monitoring purposes, with each event type that can occur on a processor assigned a different address. Each probe stores a 32 bit value into the address of the appropriate event type. The value stored is a parameter of the event. For example, a probe that records process termination might store the identifier of the terminated process into the address associated with the event type "termination of process".

The EPU is connected to the system bus, and is responsible for recognising store instructions that are done by probes, and recording the event records in a 16 element FIFO. The CPU in the TMP extracts event records from the FIFO, and is responsible for local analysis, and communication with the central monitor station. TMPs are connected to the central monitoring station and to each other by a network used only for monitoring. Each

TMP carries out a significant amount of analysis, with local measurement summaries sent to the central station at one second intervals. The central station can also query TMPs for detailed information needed to satisfy a user request.

The central monitoring station provides for global analysis, and also provides a graphical user interface to the monitoring system.

(4) JADE

Although the structure of JADE [JOYC87] is not hierarchical, JADE is still structured using layers of components. Probes are included in a special version of an interprocess communication library. Each probe sends messages to a *channel* process, with one channel process present on each node. Analysis programs in JADE are known as *consoles*, and several consoles have been developed. When a console starts executing, it registers with one or more channel programs. Each channel forwards all events that it receives to all consoles registered with the channel, and all consoles operate independently of each other. JADE is not hierarchical in structure as each channel can forward events to more than one console.

(5) Summary

The monitors discussed all conform to the general description of the characteristics of distributed monitors given in Subsection 7.2.1. All are based on several layers of monitor components, with most monitors having a hierarchical structure.

A common problem in the design of distributed monitors is how to handle very high event rates. Using a hybrid approach, such as the one adopted in TMP, events can be recorded at a very high rate, with little effect on the performance of a node. Typical event rates in experiments using TMP were 600-800 events per second, achieved with an overhead of 0.1% per node. Another technique is to filter and analyse as much as possible within each node, thus reducing the amount of information that has to be passed to the analysis tools. Also, lower level components may send some or all types of information to upper level components only on request.

7.2.3 Structure of an Interaction Network Monitor

The hierarchical structure of most existing distributed monitors, as shown in Figure 7-1, should be suitable for an interaction network monitor. Based on such a structure, the likely components of an interaction network monitor are discussed in detail in Sections 7.3 and 7.4. The design of probes, to be described in Section 7.3, is to a large extent independent of how the higher layers are structured. The event recording, analysis, and database functions are more interdependent, and are covered together in Section 7.4.

7.3 Probes

The steps in probe design are: (1) deciding which types of event should be recorded, (2) determining the locations of probes needed to record those event types, (3) deciding on the information to be recorded for each event type, and (4) adding the probes to the target software. In this section, we discuss how these four steps might be done for an interaction network monitor.

A set of probes for an interaction network monitor must be designed to supply the information needed: (i) to determine the extent of the processing performed for each task, that is the *task boundary* and (ii) to construct an interaction network for each task. To satisfy (i), probes must be installed to record every sub-task start and sub-task end event, as will be discussed in Subsections 7.3.1 and 7.3.2. To satisfy (ii), one or more sub-task identifiers must be recorded in the event record produced by each probe, as will be discussed in Subsection 7.3.3.

To deal with join events, we must make assumptions about which sub-task is the continuing sub-task. In part (6) of Subsection 4.2.4, we suggested three assumptions. Where sub-tasks that belong to different tasks join, the choice of the continuing sub-task affects task boundaries. A major goal when constructing INMON was to show that the three assumptions suggested are reasonable in practice, and that their use results in accurate determination of task boundaries. Being able to determine accurately the extent of each system reaction is obviously essential for recording interaction networks.

7.3.1 Event Type Selection

Events recorded in an interaction network fall into two categories: structural events (sub-task start events and sub-task end events), and object-use events. Important types of event are listed below, grouped into these categories. One or more probes must be installed to record events of each type.

Structural events:

1. task start
2. thread creation
3. completion of work for a sub-task by a thread
4. message send
5. message receive
6. message join/spilt

7. shared memory write

8. shared memory read

9. shared resource release

10. shared resource acquire

Object-use:

11. begin_use

12. end_use

7.3.2 Probe Placement

Identifying probe points should be straightforward for event types 2, 4-6, and 9-12, as these event types correspond to well defined events within the target software. Probe points for events of type 1, *task start*, should also be easy to identify. Because the start of each task is triggered by a user action, probe points to detect such events will be in the low level input routines that detect key-strokes, mouse movements, mouse button events, and so on.

Identifying probe points for events of type 3, where a thread finishes work for a sub-task, have proved to be more difficult to detect. If the thread terminates, then it has certainly finished working for its current sub-task, so there must be a probe that detects thread termination. That leaves the problem of detecting when a thread finishes work for a sub-task part way through the execution of the thread. To detect this point precisely may require a knowledge of the internal operation of the application that the thread is executing.

One approach, that should in practice be accurate enough, is to assume that a thread finishes working for a sub-task when it performs an operation that will result in it being associated with another sub-task. An example is where a thread receives a message. When the message is received, the sub-task that the thread was associated with terminates, and the thread becomes associated with the message sub-task.

Detecting individual events of type 7, *shared memory write*, and type 8, *shared memory read*, at the physical memory reference level requires additional hardware that can produce address traces. Extracting the information required for interaction networks from the enormous amount of information contained in an address trace would be impractical. Where shared variables are accessed using programming language constructs, probes to record events of types 7 and 8 could be added to the run-time support functions of the language.

Probes required for recording the beginning and ending of sub-tasks will most often be installed in the operating system kernel. Some object-use probes will be installed in the kernel, to record use of kernel resources like processors, while others will be installed outside the kernel, to record use of resources like procedures and database relations. Some probes will be coded by hand, while others will be inserted automatically. For example, kernel probes will usually be coded by hand, whereas procedure call and return probes will be inserted automatically by compilers.

7.3.3 Parameter Selection

Each probe, possibly in conjunction with the event recorder, will store in an event record values of *parameters* that describe the event. Parameters must be selected so that event records contain all of the performance data needed. Parameters are of two types: *event-independent parameters* and *event-dependent parameters*. Values of event-independent parameters are recorded for all events, and include the time of the event and the node on which it occurred. Event-dependent parameters are specific to types of event, and include message length for message send and receive events, and disc address for disc access begin events. An event record contains values of all event-independent parameters, and of those event-dependent parameters defined for the type of event being recorded.

Parameters are now discussed under two headings. First, we deal with those parameters that provide the information necessary to identify all event records that belong to a given task, and to construct an interaction network from those records. Techniques for maintaining the variables used as values for these parameters are also discussed. Second, we describe all other parameters, such as the event time and the network address of the node on which the event occurred.

(1) Recording the interaction network structure

To identify the event records that belong to a task, one possible method is to include a task identifier as one of the event-independent parameters. When the first event in a task occurs, a unique task identifier is generated. This identifier is subsequently recorded in the event records of all events that occur during execution of the task. This approach does identify all events that occurred in a task, but provides little assistance towards constructing an interaction network from the individual event records.

A better task identification method is based on generating unique identifiers for each sub-task whenever a new sub-task is created. All event records for events that occur in the sub-task will include that *sub-task number* as an event-independent parameter.

Some event-dependent parameters are required to record the links between the sub-tasks of a task. All fork, multi-way fork, and message join/split event records have as

event-dependent parameters the sub-task numbers of the created sub-tasks. All join and message join/split event records have as event-dependent parameters the sub-task numbers of the terminating sub-task.

The information provided by the sub-task numbers just described is sufficient to allow an interaction network to be constructed from the set of event records of a task. First, a tree of records is constructed. At the root of the tree is the record of the source event of the task. Each record in the tree has as a child the record of the next event in the sub-task, if such an event exists. Also, the record R of each fork, multi-way fork, and message join/split event has as children the records of the first events in all of the sub-tasks created at R. Second, the tree is turned into an interaction network by adding links from the last record of each "terminating" sub-task to the record of the join or message join/split event at which that sub-task terminated.

So far, we have assumed simply that the event recorder in each node has available to it the number of the sub-task in which an event has occurred. We now describe the three things that must be done to provide event recorders with this information. First, it must be possible to associate a sub-task number with each thread and with each message. Second, procedures must be in place to assign sub-task numbers to threads and to messages. Third, each event record must include in its parameters one or more sub-task numbers, as described above.

To hold the sub-task number associated with a thread, a sub-task number field can be added to the thread's state information. To hold the sub-task number associated with a message travelling across a network, a new field can be added to the message header. To hold the sub-task number of a message stored in a node, a new field can be added to the message queue data structures.

The sub-task number stored for each thread and each message must be updated whenever there is a change in the sub-task that is associated with the thread or message. The event types for which this updating is necessary are:

(1) Source events. A new sub-task number is generated, and assigned to a new message, or to the initial thread.

(2) Fork and multi-way fork events. New sub-task number(s) are generated, and assigned to the new messages, or to the new thread.

(3) Join events. The sub-task number of the continuing sub-task is assigned to the message or the thread that continues from the join. For example, in the case of a thread receiving a message, the sub-task number of the message is assigned to the thread.

(4) Message join/split event. A new sub-task number is generated and assigned to the message created at the join.

Finally, a method must be available for each node to generate sub-task numbers that are unique across all nodes of a distributed system. A widely used method for the generation of such identifiers is for a node to concatenate its network address with the current local time. This method is based on the reasonable assumption that the interval between requests on a node to create unique identifiers will always be greater than the clock tick interval.

(2) Other parameters

Parameters required in addition to those described above are now discussed. The event-independent parameters are likely to include: the type of the event, the node on which the event occurred, the thread or message with which the event is associated, and the time at which the event occurred.

A large number of event-dependent parameters might be useful. Some possibilities are

(i) object identifier for all begin_use and end_use events. Object identifiers must be recorded if states are to be determined, as described in Section 6.1.

(ii) message length for message send and receive events

(iii) id of the created thread for create thread events

(iv) total resources used by the thread for thread termination events

(v) input device, length, and data entered for user input events

(vi) error codes for many different event types

(vii) id of the procedure called for remote procedure call events

(viii) disc address and access type for disc access events

(ix) program name and arguments for program execution events

7.3.4 Probe Implementation

Usually, probes are written as calls to a subroutine within the event recorder. The event recorder subroutine creates an event record, and can insert values of some event-independent parameters, such as the current time, without reference to the calling routine. Other event-independent parameters, and all event-dependent parameters, are supplied by the calling probe. A different approach is used in TMP, a hybrid monitor. In TMP, a probe is simply one or more store instructions, with the store address being the event type, and the value stored at the store address being an event-dependent parameter. A separate processor creates event records and assigns a value to the timestamp, an event-independent parameter.

Most filtering is done at or above the event recorder level. If a call to the event recorder incurs substantial overhead, as it would if it were a system call, then each probe might check the value of a variable to decide whether to call the event recorder. Settings of these variables would be based on control information provided by higher monitor layers. The higher layers determine the flag settings from instructions given by the performance analyst.

7.4 Event Recording and Analysis

We now discuss the components of an interaction network monitor that are above the probe level. The structure proposed is hierarchic, and similar to that of most other distributed monitors. This section is divided into two subsections. In the first, event recording and filtering that occurs within a node are described. In the second, a description is given of ways in which analysis can be performed of information from all nodes in a distributed system.

7.4.1 Event Recording and Filtering Within a Node

Each node must contain monitor components that perform:

(1) initial filtering of event records.

(2) construction and buffering of event records.

(3) transmission of event record buffers to upper monitor layers.

On each node, a *node coordinator* is responsible for passing performance information upward in the component hierarchy, and control downward. There are several ways of structuring the monitoring components within a node. Three possibilities are:

(1) Each node contains a single monitor module that incorporates all of the functions described above.

(2) Each node contains two monitor modules: an event recorder and a node coordinator. The event recorder performs event record filtering, construction, and buffering, and passes event record buffers to the node coordinator.

(3) Either (1) or (2) above, plus each cluster (as defined in Chapter 2) contains an event recorder. Such an arrangement might be attractive where the main event recorder is within the kernel, and events that we want to record are also occurring within each thread while it is executing in user mode. Recording such events individually, by making a system call each time, would take too long. It would be more efficient for an event recorder within each cluster to buffer events generated by the threads within the cluster, and to transfer several event records on each system call made to the main event recorder.

Monitor modules may be implemented as part of the software of the target system, or as software running on additional monitoring hardware, or as some combination of the two. In TMP, described in Subsection 7.2.2, only the probes are executed by the target system, while all other activities are performed by dedicated performance hardware: an analysis node, a measurement processor in each node, and a network used only for monitoring purposes connecting the measurement processors and the analysis node.

The three functions identified at the start of this sub-section can be performed as follows:

(1) Filtering. Coarse filtering is usually done by event recorders to reduce the number of event records that need to be transferred to higher layers. At this level, filtering is done to remove all event records of specified types, and all object-use event records that do not refer to objects of interest. In some situations, the probes themselves perform some filtering functions, as noted above in Subsection 7.3.4. Instructions on the filtering to be performed are received from the analysis tools, based on the performance information required by the performance analyst.

(2) Buffering. Each event recorder must have access to an area in which event records can be stored, before being sent to the next layer. One or more buffers, usually of fixed size, can be used.

(3) Communicating. Buffer contents must be passed to higher layers. Where communication is off-line, buffers will be stored in a (local or remote) file for later analysis. Where communication is on-line, each buffer will be transferred as one or more messages.

Where there is more than one monitor module on each node, the modules can communicate between themselves in the following ways. An event recorder outside the operating system kernel can pass buffers to an event recorder within the kernel through a system call interface. An event recorder within the kernel and a node coordinator can communicate by placing buffers and related data structures in an area of memory accessible to both.

7.4.2 Analysis

Analysis can be performed by a *master analysis module* on a single node, or by a master analysis module in conjunction with other analysis modules, often with one analysis module on each node. Entire interaction networks and/or performance information derived from information networks can be stored in a performance database for future analysis. The performance analyst will control the monitoring system and view performance information through the user interface of the monitoring system.

Where analysis is performed on a master node, all node coordinators will send event records to that node. The overhead in doing this is substantial, and a network used only for transmission of monitoring traffic, such as that used in TMP, may be required. Entire interaction networks will be constructed at the master node, and analysis can then proceed in the ways described in Chapter 6.

Where analysis is performed in a distributed way, additional analysis modules are needed. A likely configuration is one where there is an analysis module on each node, and a central analysis module that coordinates the analysis. Each *node analysis module* will calculate performance information that applies to activities that occur on its node, and will pass the performance information to the master analysis module, which constructs the global picture.

We now describe one type of analysis that node analysis modules could do. For each sub-task performed (at least partially) by a node, the node analysis module could pass a summary of the sub-task's activities (for example, a decomposition of a sub-task's time into compound states of interest) to the master analysis module. Such a summary is likely to be much more compact than the set of event records from which it was derived. Another possibility is for the master analysis module to request information from node analysis modules, with information being requested only to determine performance information asked for by the performance analyst.

The user interface should allow the performance analyst to control the experiments performed, and to request analyses of the data collected. Examples of experimental parameters that a user might supply through the interface include:

(1) The interactions of interest, by specifying: the nodes on which interactions of interest might start, the input devices that can initiate interactions of interest, and a set of objects which interactions of interest might use

(2) The events to record, by specifying a set of objects of interest.

A user interface should, in conjunction with the analysis tools, make available performance information derived from interaction networks. A graphical user interface should be well suited to this task.

Other functions that a user interface should provide to a performance analyst are: the ability to specify that certain data should be put into the performance database, and the ability to analyse data in the performance database. The performance database should be able to store raw data, individual event records for instance, as well as derived data, the summary of a critical path for instance. The performance database is not restricted to information on interaction networks, with information on resource utilisations, workload, and counts of events such as page faults able to be incorporated. Use of such data is

illustrated by the query "Produce a critical path decomposition for all interactions that executed during a period where average CPU utilisation exceeded 90%".

Most data in a performance database will have an associated timestamp, or time period. It would be useful if the DBMS that manages the performance database contains *temporal database* features, whereby the notion of time is built in to the DBMS and its query language [SNOD86]. Snodgrass' work on performance monitoring of distributed systems [SNOD88] utilised TQuel [SNOD87], a query language for temporal databases, to manipulate data in a performance database.

7.5 Overview of INMON

The discussion in Sections 7.3 and 7.4 suggested a monitor can be designed for recording and analysing interaction networks. To show that interaction networks can be recorded, a prototype monitor has been constructed. We now introduce that prototype, *INMON* (Interaction Network MONitor), describing first the environment for which INMON was constructed and then its structure. Chapters 9 and 10 contain a detailed description of this monitor.

7.5.1 Environment

INMON was constructed to operate on a network of Sun workstations in the Department of Computer Science at the University of Canterbury. When this project was started, the departmental Sun network consisted of five Sun 3 workstations connected by a 10Mbps ethernet.

The system clock in Sun 3/50s and 3/60s has a resolution of 20ms. This resolution was found to be too coarse, so clocks with microsecond resolution were installed in two of the Sun 3s. Section 8.1 contains more detail on these clocks.

Because the construction of INMON required extensive modifications to the operating system kernel, the source code for version 4.0 of SunOS was purchased. Version 4.0 does not run on Sun 4s, so work with INMON has been restricted to the Sun 3s.

Using the classifications of operating systems introduced in Section 2.2, SunOS is closest to a network operating system. SunOS was initially the 4.2 BSD Unix operating system, with some network services added. The Sun Network File System (NFS) [SAND85] is the most important network service. NFS, and many other Sun network services, use remote procedure call for communication between clients and servers.

7.5.2 Structure of INMON

The structure of INMON is now described. The main aims behind the design are discussed first, as they had significant influence on design decisions. The structure of INMON is then dealt with in two parts: first, the probes and the event recorder and, second, the analysis tools.

(1) Aims

The main aim behind the implementation of INMON has been to show that interaction networks can be identified and recorded. Less emphasis has been placed on other aims. For example, an event recording mechanism with low overhead, or a user friendly interface, were not high priorities.

Such considerations are important for a production quality monitor, but were outside the scope of INMON. Solutions for these problems have been implemented in many existing monitors. For example, TMP provides a very efficient event recorder capable of recording events at a high rate, and IPS-2 provides a sophisticated user interface. The author feels that such techniques can be applied in the construction of a production quality interaction network monitor.

(2) Probes and event recording

In INMON, all probe points selected are in the kernel. A system call is provided so that programs running outside the kernel can record events, but to date little use has been made of it. The INMON event recorder is implemented in the kernel. Each probe calls a function within the event recorder. When called, the event recorder function constructs an event record and adds the event record to a buffer. A background process copies buffers of event records to a log file, which may be a local file, or a remote file accessed using NFS. One log file is produced by each Sun, so the analysis tools are responsible for constructing interaction networks from event records stored in many log files.

The INMON probes and event recorder are described in Chapter 9.

(3) Analysis

Several analysis tools have been developed, including programs to:

(i) provide a text dump of the contents of a log file.

(ii) extract interaction networks from a group of related log files.

(iii) display an interaction network.

(iv) provide some analysis of an interaction network.

(v) provide a graphical interface to the functions performed by (i), (iii), and (iv) above.

The tools that perform analysis on interaction networks (tools (iii), (iv), and (v)) take as input a file containing all of the event records of a task that was produced by tool (ii), so analysis is centralised rather than distributed. The analysis tools are described in Chapter 10.

7.6 Summary

Many monitors for distributed systems have a hierarchical structure, which is appropriate for a monitor that is to record data needed to construct interaction networks. At the bottom level of the hierarchy are the event probes. Probe placement for an interaction network monitor was discussed, as were the parameters whose values will allow an interaction network to be constructed from the event records of a task. Event records are passed to an event recorder, and from there to the analysis components of the monitor. These analysis components may operate in a centralised or a distributed manner.

INMON, the author's prototype monitor, has been introduced. The main aim behind development of INMON, has been to demonstrate that software can be written to record and analyse interaction networks. Successful construction of INMON has shown that it is possible: to identify the set of event records that make up each task and, from there, to construct and display interaction networks.

INMON consists of: a set of probes on each node, an event recorder on each node, and a set of analysis tools that operate in a centralised fashion. INMON is described in detail in Chapters 9 and 10, and experiments performed using INMON are described in Chapter 11.

Chapter 8

Timing Issues

Accurate recording of time is important in any system for performance measurement. For loosely coupled distributed systems, there are two main timing issues:

1. The resolution of the clock in each node. The resolution of the Sun 3 system clock was found to be insufficient for our purposes. Section 8.1 describes how this problem was solved.

2. Clock synchronisation. Each node has a local clock, and events are timestamped by these clocks. If clocks are not kept tightly synchronised, then it is not possible to measure accurately the interval between a message send and a message receive, and it is even possible for the timestamp recorded for a message send event to be greater than the timestamp recorded for the corresponding message receive event. The clock synchronisation problem, and solutions to it, are discussed in Section 8.2.

A high degree of clock synchronisation is needed for the recording and analysis of interaction networks, as described in Section 7.1. We intended initially to use some existing clock synchronisation program to provide a global timebase for INMON. Since none of the programs readily available achieved a small enough maximum synchronisation error, the clock synchronisation problem was investigated in detail and a new clock synchronisation program was developed. This program, `uoctimed`, is examined in detail in Section 8.3, and experimental results from use of the program are summarised in Section 8.4. Some changes to `uoctimed`, which address problems highlighted by the experimental results, are described in Section 8.5. A summary is given in Section 8.6.

8.1 Clock Resolution

In Sun 3 systems, clock interrupts occur at a rate of 100 Hz, with every second interrupt being ignored. Therefore the system clock is updated 50 times per second, giving a resolution of 20ms. This resolution is not sufficient for accurate timing of CPU burst lengths, message delay times, disc I/O waits, and many other things that we are interested in measuring.

Danzig and Melvin [DANZ90] also found the low resolution of the Sun 3 system clock to be a problem. To solve that problem, they designed a high resolution clock which plugs

into the Sun 3 DES chip socket, a socket unused on most Suns. The Danzig-Melvin (D-M) clock has a maximum resolution of 250ns, and was used by Danzig for measuring the components of the time between when a UDP packet is received by a node, and when it is received by the destination process on that node [DANZ91].

For our project, D-M clocks were installed in two Sun 3/50s: weka and tui. Support software, including a device driver to be added to the SunOS kernel, is supplied with each clock. The kernel device driver includes the new system call `gettmr` which returns the current D-M clock time. To use the D-M clock for recording of interaction networks, the following things were done:

(1) The D-M clock was used in "timeval" mode, in which the clock operates with a one microsecond resolution. When read, the clock returns a 64-bit value in the form of a Unix timeval structure consisting of two 32-bit integers: the elapsed seconds and microseconds since 00:00 GMT, January 1, 1970 (zero hour). A simple program, `utimeset`, was written to put a D-M timer into timeval mode and then set it to the current system time. Both weka and tui execute `utimeset` at system boot time.

(2) Two further system calls were added. The `settmr` system call sets the D-M clock to the time specified. The `adjtmr` system call is used for clock synchronisation and is discussed later in the chapter.

(3) The interaction network event recorder uses the D-M clock to timestamp events.

Unix manual pages for `adjtmr`, `gettmr`, `settmr`, and `utimeset` are given in Appendix B.

8.2 Clock Synchronisation

We now introduce the clock synchronisation problem. The reasons for keeping clocks synchronised in a distributed system are discussed in Subsection 8.2.1. An approach that can be used for event record timestamps, as outlined in Subsection 8.2.2, is not to synchronise clocks, but to correct timestamps during data analysis. Another approach, discussed in Subsection 8.2.3, is to determine a partial ordering of events by including timestamp information in all messages. Other approaches, described in Subsection 8.2.4, synchronise clocks. Communication protocol support for clock synchronisation is outlined in Subsection 8.2.5, with some clock synchronisation programs surveyed in Subsection 8.2.6.

8.2.1 Motivation for Clock Synchronisation

Synchronisation of clocks in a distributed system is attempted for many reasons. One is to provide a global ordering for events that occur in a distributed system. Lamport

describes one way of imposing a global ordering on a set of events occurring in a distributed system, and then goes on to describe a distributed mutual exclusion algorithm based on the global ordering of events [LAMP78].

Many authentication mechanisms used in distributed systems require clock synchronisation across the set of nodes on which they run. Timestamps are used to help prevent encrypted messages being replayed at a later time by an intruder. Each encrypted message contains time information which the receiver validates against its local clock. Kerberos is a service that enables network applications to identify their peers, by issuing encrypted tickets to applications [KOHL91]. In version 5 of the Kerberos protocol, the period for which a ticket is valid is included within the ticket.

In the secure authentication option of Sun's RPC [SUN88b], a client and a server use a session key supplied by the client in the first remote call. In every call that a client makes, included in the message sent by the client is the client's local time encrypted with the session key. To authenticate the client on each call, the server decrypts the client timestamp with the session key, and checks that the client timestamp is greater than the previous client timestamp, and less than the server's current local time plus a period specified by the client in the first call. Clearly, both authentication mechanisms rely on some degree of clock synchronisation.

The clocks of all nodes providing a file service in a distributed system should also be synchronised, so that times associated with file accesses closely approximate the true access order. Consider `make` [FELD79], a utility designed primarily to automate the process of constructing executable programs from program source files. Using `make` it is possible to declare that an executable file depends on a program source file, and that to create the object file the source file is to be compiled using, say, a C compiler. `make`, when run, checks to see if the last modification time of the source file is later than that of the object file, and if this is the case the source is recompiled to produce the object file.

Consider a case where the source file and executable file are stored on different nodes, and the clocks on the nodes are not synchronised. If the clock on the node storing the source file is set to a time later than that of the clock on the node storing the object file, then it is possible for `make` to decide that a compilation is required when in fact it is not. If the clock on the node storing the source file is set to a time earlier than the clock on the node storing the object file, then it is possible for `make` to decide that a compilation is not required when in fact it is.

Finally, clock synchronisation is essential for performance measurement of distributed systems. The timestamp of a message receive event must be later than the timestamp of the corresponding message send event, so some degree of clock synchronisation is required between the sender and the receiver. If the time that a message spends in transit between

sender and receiver is to be accurately estimated, then the sender and the receiver must be synchronised more closely than if we wanted to ensure only that a message receive timestamp is always later than its send timestamp.

Others have noted the importance of clock synchronisation in the monitoring of distributed systems. Snodgrass [SNOD88] states that Lamport's partial order algorithm [LAMP78] provides the partial ordering of events required when recording events for distributed debugging. He notes also that clock synchronisation is likely to be done for the other reasons outlined in this subsection, and suggests that performance measurement tools could make use of these synchronised clocks. Also, Haban and Wybranietz [HABA90] describe clock synchronisation techniques used in the INCAS monitoring system. Results are computed at regular intervals (by default every second), with clock synchronisation used to align the intervals to the same begin and end times over all nodes.

For the purposes described above, the maximum synchronisation error that is acceptable varies considerably. In the case of Version 4 of the Kerberos authentication protocol, ticket lifetimes are multiples of 5 minutes, so synchronisation errors of tens of seconds would be acceptable. Messages passed across an ethernet may experience a delay in the order of 10^{-3} seconds. For message timings to be accurate, or even for send and receive events to be correctly ordered, the maximum synchronisation error must be well below one millisecond.

8.2.2 The Correction Approach

If a monitor collects data for off-line analysis, it is possible to timestamp events using unsynchronised clocks, then correct the local timestamps to a global timebase as part of the analysis. A simple type of correction can be calculated from estimates of the time taken to send each message, based on the message length and on statistics on message delays between the two hosts in question.

A more sophisticated method is described by Duda et al [DUDA87], in which pairs of send and receive timestamps are treated as realisations of two random variables having functional linear dependence. That is, it is assumed that the time difference between two crystal controlled clocks is a linear function of time. A least-square regression analysis is used to estimate the *time offset* and the *time offset rate* between two local clocks. The time offset is the time difference between two clocks, and the time offset rate is the rate at which the times of two clocks are drifting apart.

A convex hull analysis is given as an alternative to the least-square regression analysis. Given the initial time offset and the time offset rate, the timestamps of one node can be scaled to be consistent with the timestamps of the other node. The method is initially

developed for two sites, with suggestions of how the method might be extended to cope with N sites.

Methods that use corrections have the advantage that they impose no overhead during data recording.

8.2.3 Partial Ordering

For some applications, knowing the partial ordering of events is sufficient. Lamport [LAMP78] introduces the "happened before" relation -> defined for a set of events to be:

(1) If *a* and *b* are events on the same node, and *a* occurs before *b*, then *a* -> *b*.

(2) If *a* is the sending of a message by one node and *b* is the receipt of the same message by another node, then *a* -> *b*.

(3) If *a* -> *b* and *b* -> *c* then *a* -> *c* (-> is transitive).

Intuitively, if *a* -> *b* then event *a* may influence event *b* in some way, because either *a* and *b* occurred on the same node and *a* occurred before *b*, or *a* occurred on some other node with a message from the node on which *a* occurs being sent after *a*, arriving (directly or indirectly), at the node on which *b* occurs, before *b* occurs.

The -> relation is important in debugging [SNOD88]. If an error is observed at the time of event *b*, then any event *a* such that *a* -> *b* might represent the initial error which resulted in an error being observed at *b*. Also Lamport [LAMP78] describes a mutual exclusion algorithm based on the partial order provided by the -> relation.

Lamport describes an algorithm that maintains a logical clock at each node, so that if *a* -> *b* then the logical time that *a* is timestamped with is less than the logical time that *b* is timestamped with. The algorithm requires the following:

(1) A node must advance its logical clock between events.

(2) When a message send event occurs, the logical clock time of the sending node is sent as part of the message. When the message is received, the logical clock of the receiving node must be advanced past the message timestamp, if this is not already the case.

The algorithm imposes some extra overhead through the addition of a timestamp to each message. The algorithm is not suitable for use in measuring message transmission times, because logical clocks do not measure physical time.

8.2.4 Clock Synchronisation Solutions

Clocks can be synchronised using hardware or software methods. In hardware methods, clocks are synchronised using additional hardware that usually includes a separate network to allow the clocks to communicate [KRIS85]. Such solutions can provide very tight synchronisation, but the cost of the additional hardware may be high. Another method is to synchronise clocks to radio signals broadcast from a satellite or local transmitter. Either of these synchronisation methods would have provided adequate synchronisation, but neither was available. Only software solutions are considered further.

Several clock synchronisation algorithms have been developed to keep physical or logical clocks synchronised, by calculating corrections to local clocks through exchanging messages containing local clock readings. A clock synchronisation algorithm is distributed across each node of a network, and executes four functions in sequence [VERV91]:

- determination of the *resynchronisation moment*,
- determination of (some) *remote clock values*,
- estimation of *global time*, and
- adjustment of the *local clock*.

Local clocks run freely between adjustments. The time between adjustments is the resynchronisation interval. In the following sections, the four functions outlined above are described, as well as the issues of synchronising with a time standard, fault tolerance, and error bounds.

(1) Determining the resynchronisation moment

Two different approaches, global and distributed, have been identified to the question of deciding when clocks should resynchronise [VERV91]. In the global approach, a master node broadcasts a message to all other nodes to inform them that they should perform a resynchronisation. In the distributed approach, each node decides when to resynchronise based on local information. All nodes may agree on a time to resynchronise, and look for that time on their local clocks. Alternatively, each node may decide when to resynchronise individually and independently of all other nodes.

(2) Determining remote clock values

Readings of remote clock values are used to determine the differences between a node's local clock and the clocks of other (remote) nodes. A common way to calculate the clock difference between two nodes, discussed in (i) below, is for request and reply messages to be exchanged. Other methods are discussed in (ii).

(i) The difference between the local clock and a remote clock can be calculated from the exchange of two messages. The difference can be estimated based on local timestamps before the request is sent and after the reply is received, and one or more remote timestamps in the reply.

One possibility is as follows. Node N_A records its current time (t_1) then sends a message to Node N_B. Node N_B receives the message from Node N_A and replies with a message containing the time at Node N_B (t_2). When Node N_A receives the reply it records its local clock time (t_3). Node N_A calculates the correction *c* needed to synchronise its local clock to Node N_B by:

$$c = t_2 - (t_1 + t_3) / 2 \qquad (8.1)$$

This formula is the basis of methods used in TEMPO [GUSE89], by Haban and Wybranietz [HABA90], by Vervoort *et al* [VERV91], and by Cristian [CRIS89]. The formula estimates the time at which Node N_B records t_2 to be mid-way between t_1 and t_3, and is based on two assumptions. The first is that messages between nodes in opposite directions will (on average) experience an equal delay on the network. This assumption is reasonably valid for bus networks such as ethernet, but not for ring networks where some adjustment is needed according to the relative positions of the nodes within the ring. Accuracy of the formula therefore depends on the variability of network delay.

The second assumption is that timestamp t_2 is midway between the time at which Node N_B receives the incoming message and the time at which Node N_B sends its reply. If this assumption is not reasonable, then a better approximation can be obtained if Node N_B records two times: the time at which it receives the message from Node N_A and the time at which it sends the reply to Node N_A. The correction to be applied to the clock in Node N_A is then half the difference between the recorded message delays. This method is used in the Network Time Protocol (NTP) [MILL88b]. In NTP, nodes exchange messages regularly. The receiving node calculates a correction every time it receives a message, then sends a message containing two timestamps back to the sender, which uses these to calculate a correction, then replies with a message containing two timestamps, and so on.

A subset of the algorithms based on pairs of request and reply messages are algorithms that use *probabilistic clock synchronisation* [CRIS89]. In such algorithms, a node will keep sending request messages until it receives a reply such that the delay between sending a request and receiving the corresponding reply, the message *round trip time (rtt)*, is close to the lowest delay, the *minimum rtt*, expected for the network connection used. We refer to this "close enough" value as

the *threshold rtt*. Where the threshold rtt is very close to the minimum rtt, clock differences calculated from replies whose rtt is less than the threshold rtt are very accurate, as the message delay variation is minimal. This technique can give very accurate synchronisation most of the time, at a potentially high communication cost. The accuracy of probabilistic algorithms and of other (deterministic) algorithms is discussed later in this section.

(ii) Other clock synchronisation methods determine a correction based on the passing of a single message rather than on the exchange of two messages. For these methods, the message delay must be estimated. Lamport [LAMP78] describes a clock synchronisation algorithm which adds a minimum message delay to any timestamp received to make it comparable with the current local time. The minimum delay is ≥ 0 and less than the total delay of any message. De Carlini and Villano [CARL88] describe a clock synchronisation algorithm designed for a transputer network. The nodes whose clocks are being synchronised are arranged (logically) in a ring, with synchronisation messages flowing in one direction around the ring. The message delay for a single step around the ring is estimated to be the total time for the last synchronisation message to travel around the ring divided by the number of nodes in the ring.

Ramanathan et al [RAMA90a] describe a clock synchronisation technique for a partially-connected network of nodes, where messages from one node to another may be relayed by one or more intermediate nodes. Their method for calculating clock differences relies on an additional piece of hardware which adds the current local time to each synchronisation message as it is sent and as it is received. Using this hardware, we can accurately record the initial send and final receive times, and from these time calculate the amount of time a message is delayed at an intermediate node. The variations in network transmission times are apparently ignored, but these should be small compared to the delay variations eliminated by this technique, which are the initial sending delay (time from when the initial send is requested until the message is sent), intermediate node delays, and the final receive delay (delay from when the final message is received until it is actually processed).

(3) Estimating the global time

Obtaining one or more estimates of the difference between local clock time and that of one or more remote clocks is the first step in arriving at a correction to be made to the local clock. The second step is determining, from these estimates, the global time, with the correction to make to the local clock being the difference between the global time and the current local time.

Clock synchronisation algorithms are designed either to synchronise all clocks with one particular master clock, or to synchronise to a "network time" whose value is based on readings of some or all of the clocks. The computation of network time can be performed by a master node, or may be performed using a distributed algorithm.

Where all clocks in the system are synchronised to a designated master clock, the only estimate relevant for Node N_A is the difference between its clock and the master clock, with each correction based on one or more estimates of this difference. De Carlini and Villano [CARL88] describe a network of transputers which synchronise to a master clock by communicating clock synchronisation information around a logical ring. Each correction is based on one difference estimate.

Cristian [CRIS89] describes an algorithm where slaves synchronise to a one of a set of masters, with a probabilistic algorithm used to determine differences between the slave and master clocks. Each master is synchronised to a reliable external source of time.

In the TEMPO system [GUSE89], one of the nodes acts as a master (but not as the master clock), and all other nodes act as slaves. The master estimates the difference between its own clock and every other clock using a technique similar to Cristian's probabilistic algorithm. For each slave, if no replies are received by the master within the threshold rtt after a fixed number of attempts, the difference estimate is calculated from the request and reply messages with minimum delay (in Cristian's algorithm this situation is handled by the slave leaving the set of synchronised clocks). The master ignores any clocks which differ from a majority of the others by more than a small amount. It averages the remaining differences, and adjusts its own clock by this amount, and then informs each slave of the correction to apply to its clock.

(4) Applying corrections

Synchronisation algorithms may adjust clock values. An alternative to modifying the clock itself is for each node to maintain an offset value, showing the estimated difference between the local clock and the global time. In effect, one introduces a logical (that is, a software) clock. Haban and Wybranietz [HABA90] use this approach. Maintaining offsets has the advantage that errors due to the time taken to adjust the clock are eliminated.

If it is necessary that clock readings be monotonically increasing, then setting a clock to an earlier time can be achieved by slowing the clock. TEMPO uses the `adjtime` system call provided by Berkeley Unix for this purpose [GUSE89]. Cristian's solution to this problem is to make the offset a linear function rather than a constant [CRIS89]. The offset function is recalculated at each synchronisation so that the logical clock time at the time of the adjustment remains the same, with the adjustment *amortized* over the resynchronisation interval.

Vervoort *et al* [VERV91] introduce the idea of *rate adjustment*, where the rate of a local clock is adjusted so that future adjustments will be smaller. Rate adjustment is possible where logical clocks are being synchronised, but is usually not possible for hardware clocks.

(5) Synchronisation with a time standard

Some clock synchronisation algorithms try to keep the clocks in a distributed system synchronised with each other, and synchronised with an external time standard, usually Universal Time Coordinated (UTC), formerly Greenwich Mean Time (GMT). Implementations of NTP [MILL88b], [MILL89a], [MILL90b] ensure that a large sub-set of the hosts in the DARPA Internet synchronise their clocks to UTC. In NTP, a small number of primary servers (of the order of 16 [MILL90b]) are synchronised by radio or satellite to national time standards. Other hosts are arranged in hierarchies in such a way that they all synchronise, either directly or indirectly, with one or more primary servers.

Other algorithms are not designed to keep clocks synchronised with an external time standard, although usually clocks are set initially to a time close to UTC.

(6) Fault Tolerance

The ability to deal with faults may be important. One class of clock synchronisation algorithms is designed to be fault-tolerant in the presence of *Byzantine faults*. In the Byzantine fault model [LAMP85], a clock can exhibit any fault behaviour including being two-faced, that is, lying to other nodes about the clock's value. Ramanathan *et al* provide a good overview of clock synchronization algorithms designed to operate in the presence of Byzantine faults [RAMA90b].

The Byzantine fault model is a worst case, and algorithms designed to operate in the presence of Byzantine faults incur significant overhead. Many clock synchronisation algorithms, though they may not deal with Byzantine faults, incorporate at least some level of fault tolerance. With TEMPO, for example, clock readings are discarded if they differ from the majority of the other clock readings by more than a small amount. Also, if the node acting as the master fails, the remaining nodes elect a new master. Any algorithm that has a master node of some sort requires an election procedure if it is to be fault-tolerant. NTP implementations perform careful filtering of difference estimates used in performing synchronisation. Also, hierarchies of NTP clock servers are tolerant to the loss of nodes and communication links.

(7) Error bounds

The accuracy achievable by clock synchronisation algorithms is determined by a number of factors, the most important being:

(i) Clock drift. Since local clocks run freely between resynchronisations, the relative drift rates of two clocks will determine the rate at which their times diverge between resynchronisations.

(ii) Resynchronisation interval. The longer the interval between clock synchronisations, the more two clocks will drift apart before being resynchronised. An upper bound for the drift between clocks over the resynchronisation interval is given by 2pW, where p is an upper bound for drift rate of all clocks being synchronised, and W is the length of the resynchronisation interval.

(iii) Network delay variability. The accuracy of all methods of determining the difference between a remote clock and a local clock deteriorates with increasing variability of network delay. This is because determining a difference involves comparing local and remote timestamps recorded before and after message(s) are exchanged. Adjustments for network delays must be made to one or more timestamp(s) to make the timestamps comparable. As the variability of the network delay experienced by messages increases, so the accuracy of the network delay adjustment decreases, resulting in decreasing accuracy of difference estimates.

While most authors state that the network delay variability is a factor in determining the accuracy of clock synchronisation, what is actually important is the variability of a somewhat longer time, that is the time from when the sender records its timestamp to the time at which the receiver timestamps the message containing the sender's timestamp. Network delay variability is a component of this, but delays in performing send and receive processing on sender and receiver are also important.

Lamport [LAMP78] derived the following error bound for one particular clock synchronisation algorithm:

$$\text{error} < d(2pW + E) \qquad (8.2)$$

where p and W are as above, E is an upper bound on the unpredictable delay of a message, and d is the network diameter. For an ethernet, which is a broadcast bus network, d = 1. This formula could be extended to incorporate a number of lesser factors, such as the resolution of the clocks to be synchronised, and the error involved in reading and setting clocks.

8.2.5 Protocol Support

Many network protocols provide message formats that can be used for clock synchronisation purposes. Mechanisms provided by the DARPA Internet protocols are now briefly discussed.

(1) Internet Protocol

The internet protocol (IP) provides the ability to collect timestamps in the header of an IP datagram [POST81a]. If the timestamp option is included in an IP datagram then each gateway through which the datagram travels en route from the source host to the destination host adds a timestamp (in milliseconds since midnight UTC). As the source host time is not recorded, the IP timestamp option is not very useful for clock synchronisation .

(2) Internet Control Message Protocol

The internet control message protocol (ICMP) includes the message types: timestamp request (TSTAMP), and timestamp reply (TSTAMP_REPLY). Each of these message types contains three timestamp fields: the *originate* timestamp, the *receive* timestamp, and the *transmit* timestamp. All timestamps are 32 bit integers that contain a timestamp in units of milliseconds since midnight UTC. ICMP provides the timestamp request and reply messages to allow two nodes to synchronise their clocks [COME88].

ICMP timestamp messages are exchanged in the following way. Node N_A sends a TSTAMP message to Node N_B. The *originate* timestamp of the TSTAMP message is set to the time at which Node N_A sends the TSTAMP message. The other two timestamps in the TSTAMP message are not used. When Node N_B receives a TSTAMP message from Node N_A it responds with a TSTAMP_REPLY message. The *originate* timestamp of the TSTAMP_REPLY is copied from the TSTAMP message. The *receive* timestamp is set to the time at which Node N_B receives the TSTAMP message, and the *transmit* timestamp is set to the time at which Node N_B sends the TSTAMP_REPLY. When Node N_A receives the TSTAMP_REPLY it records the time of arrival. A correction can then be calculated from these four timestamps (the three in the TSTAMP_REPLY plus the time at which Node N_A received the reply), by halving the difference between the message delays.

In TEMPO [GUSE89], the master node uses ICMP to get timestamps from remote clocks. The master stores the current time as the *originate* timestamp in the TSTAMP message, and then makes a system call to have the kernel send the TSTAMP message to a slave. Assuming the slave node is running Berkeley Unix, the kernel of the slave node receives the TSTAMP message, turns it into a TSTAMP_REPLY message, sets both the *receive* timestamp and the *transmit* timestamp to the current time, then sends the

TSTAMP_REPLY back to the master. The slave processing occurs entirely within the kernel. When the master receives the TSTAMP_REPLY it records the time. As the slave has returned a single time the master uses equation 8.1 to calculate the difference between its own clock and that of the slave.

(3) Daytime and time protocols

The daytime protocol [POST83a] and the time protocol [POST83b] are simple protocols that allow one machine to request the time from another. Both can be used via either the UDP [POST80] or TCP [POST81c] protocols. In the daytime protocol, the time is returned as an ASCII string for which there is no fixed format. In the time protocol, the time is returned as a 32 bit integer containing the number of seconds since January 1 1900 GMT.

The time protocol could be used in a clock synchronisation algorithm, but its one second resolution limits the situations in which it might be useful. The daytime protocol is not so useful given that there is no fixed format for the ASCII string that it returns.

(4) Network Time Protocol

The Network Time Protocol (NTP) is a protocol designed expressly for the purpose of synchronising clocks in a large internet. Timestamps in NTP messages are stored in a 64 bit fixed point format, in seconds relative to 0000 UTC on 1 January 1900. The integer part is in the first 32 bits and the fractional part in the last 32 bits. The precision of this representation is about 0.2 nanoseconds. NTP messages contain *originate*, *receive*, and *transmit* timestamps which have the same meaning as in ICMP. Clock differences can be calculated in the same way as they are for ICMP.

NTP is a complex protocol which has gone through several revisions. Xntp, a Unix implementation of version 2 of the NTP protocol [MILL89a], is widely available. Version 3 of NTP is being developed.

8.2.6 Measured Accuracy

The accuracy achieved by any clock synchronisation algorithm can be expressed as the largest difference between any two of the clocks being synchronised. Usually an upper bound on this difference and/or the average difference are reported. For TEMPO in an ethernet LAN environment, Gusella and Zatti [GUSE89] calculated the upper bound on the synchronisation error to be 30ms, with the mean error estimated to be between 18 and 20ms. They reported that differences of 25ms or more were rarely observed.

Mills [MILL90b] describes experiments to determine the accuracy of NTP clock synchronisation in the DARPA Internet. It was found that 30% of clocks were accurate to within 30ms of UTC, and over 90% were accurate to within 1s of UTC.

De Carlini and Villano [CARL88] report maximum errors of less than 100μs in measurements under various workloads of a synchronisation algorithm developed for a transputer network. Ramanathan et al [RAMA90a] calculate error bounds of hundreds of μs for hypercube and mesh networks. Their algorithm proposes additional hardware to assist accurate timestamping of message send and receive events, as described in Part (2) of Subsection 8.2.4. Haban and Wybranietz [HABA90] measured an average clock synchronisation error in the order of 100μs for their clock synchronisation implementation, but they had the advantage of a LAN used only for passing performance and clock synchronisation information.

Cristian [CRIS89] estimates maximum errors of the order of 1ms for an algorithm based on probabilistic clock synchronisation.

8.2.7 Example systems

Aspects of many clock synchronisation implementations have been discussed. Two widely available clock synchronisation algorithms are now described.

(1) TEMPO

TEMPO [GUSE89] was developed with the aim of synchronising the clocks of a collection of nodes running Berkeley Unix, and connected by a LAN. TEMPO, part of the system software of Berkeley Unix 4.3 BSD, will run under BSD compatible operating systems like SunOS, and can be easily obtained by anonymous ftp.

Each node runs a `timed` (time daemon) process, with one `timed` the master, and the others slaves. If the current master becomes unavailable the slaves elect one of their number to become the new master. The master initiates a clock synchronisation round every 4 minutes. It computes the difference between its own clock and that of each slave using the ICMP protocol and equation 8.1. Any differences which seem likely to be faulty are discarded, and the remaining differences are used to calculate the network time. The master then adjusts its own clock, and sends a message to each slave informing it of the amount by which the slave node's clock should be adjusted.

(2) Network Time Protocol (NTP)

NTP [MILL88b], [MILL89a] was developed with the aim of synchronising clocks to UTC in a large and widely distributed internet, namely the DARPA Internet. Mills

[MILL90b] describes an experiment which showed 789 nodes using NTP in the DARPA internet alone.

In NTP, each node is associated with a stratum indicating how far away it is from a node with an external source of UTC. Nodes in stratum 1 have an external source of UTC, usually provided by radio clocks or signals from the GEOS satellite system. Mills [MILL90b] is aware of at least 16 nodes in stratum 1. Nodes in stratum 2 communicate directly with nodes in stratum 1, and so forth.

While nodes in stratum 1 get their time from an external source, the other nodes synchronise their clocks based on information from one or more lower strata nodes, and zero or more nodes in the same stratum. Each node synchronises primarily to one node, and uses the information from the other (non-primary) nodes for cross checking. The node expected to provide the best synchronisation is selected as the primary node. If the primary node becomes unavailable, or is deemed to be less accurate than an alternative, then a new primary node is selected. Calculation of clock differences in NTP is described in part (2) of Subsection 8.2.5.

8.3 Implementation

We now describe `uoctimed` (University of Canterbury Time Daemon), the clock synchronisation program developed to enable interaction network recording. The major design goal for `uoctimed` was that it should synchronise a set of nodes closely enough to enable accurate measurement of network message delays. Given that ethernet messages can take around a millisecond to travel from UDP layer to UDP layer, we decided to aim for a maximum error of around 100μs. TEMPO and NTP, clock synchronisation algorithms readily available for our environment, could not be used as they have maximum errors in the order of tens of milliseconds, as described in Subsection 8.2.5.

`uoctimed` is based on a somewhat simplified version of the clock synchronisation algorithm described by Cristian [CRIS89]. A number of worthwhile improvements have, however, been made to Cristian's algorithm. These improvements are highlighted in the description of `uoctimed`, and are summarised in Section 8.6.

This section describes version 2.1 of `uoctimed`, developed following experiments with an earlier version. Version 2.2, incorporating some improvements, is described in Section 8.5

An overview of the organisation of `uoctimed` is given in Figure 8-1. A group of nodes are to have their clocks synchronised by `uoctimed`. One of them is designated as the master, and is the node with which the other (slave) nodes synchronise. The master runs

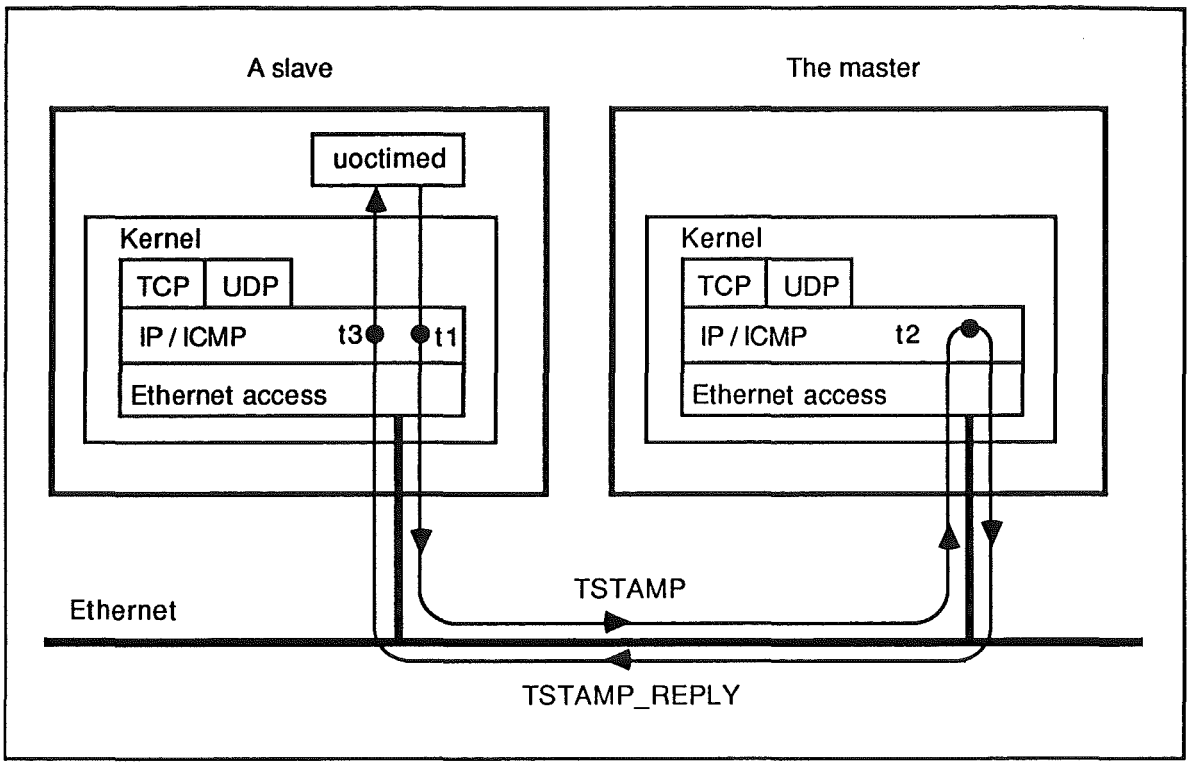


Figure 8-1: Overview of `uoctimed` architecture

no additional software, as all it has to do is to respond to ICMP TSTAMP messages, an activity performed by the kernel. Every other node runs `uoctimed`.

`uoctimed` version 2.1 is now described in detail under the same headings as those used in Subsection 8.2.4. A Unix manual page for `uoctimed` is given in Appendix B.

8.3.1 Determining the Resynchronisation Moment

Each slave determines its moment for resynchronisation individually and independently of the master and any other slaves. By default, each slave computes and applies a correction to its clock every five seconds. The default can be overridden using a command line option.

For some clock synchronisation algorithms, nodes exchange many messages when computing a correction, and exchange few, if any, messages between resynchronisations. The result is bursts of communication for resynchronisation purposes, with periods of heavy workload on the network. Since, with `uoctimed`, each slave operates independently of every other slave, the network workload from resynchronisation messages is likely to be evenly spread.

8.3.2 Determining Remote Clock Values

Clock differences are determined by a method adapted from the one used in TEMPO. This method makes use of the TSTAMP and TSTAMP_REPLY message types of the ICMP protocol [POST81b]. Node N_A sends to Node N_B a TSTAMP message, timestamped with t_1, the time at which Node N_A sent the message. Node N_B turns the TSTAMP message into a TSTAMP_REPLY, adding timestamp t_2, the time at which it handled the TSTAMP message. When Node N_A receives the TSTAMP_REPLY it records the current time, t_3. The correction required can then be calculated using equation 8.1. Figure 8-1 shows where the three timestamps are recorded.

There are two major differences in detail between the TEMPO correction calculation and that used in `uoctimed`. First, the standard ICMP timestamps (as used in TEMPO) are 32 bit integers containing the number of milliseconds since midnight UTC. To meet our goal that errors be less than 100μs, greater precision is needed.

To read the current time from a D-M clock takes approximately 20μs on a Sun 3/50, so we decided to use timestamp units of tens of microseconds since midnight modulo four hours. Because such timestamps are not in the standard ICMP units, the most significant bit of each of these timestamps must be set to 1 in TSTAMP and TSTAMP_REPLY messages [POST81b].

Given that the timestamp units are modulo four hours, all clocks to be synchronised must be initially within two hours of each other, as when comparing two timestamps the difference Dt is taken to be in the range $-2 < Dt < +2$ hours. This way of comparing two timestamps is also used in TEMPO (where timestamps are milliseconds since midnight) and X windows [SCHE90] (where timestamps are in units of milliseconds, with timestamps wrapping approximately every 49.7 days).

`uoctimed` also differs from TEMPO in that, under TEMPO, timestamps t_1 and t_3 are recorded by the `timed` process, which is outside the kernel. Under `uoctimed`, t_1 and t_3 are recorded by the ICMP layer within the kernel, so that all timestamps (t_1, t_2, t_3) are recorded by the same software layer, as shown in Figure 8-1. The major advantage is that timestamp t_3 is recorded by the interrupt handling code for receiving an ICMP message whereas, in TEMPO, t_3 is recorded in a user process, which will execute some time after the arrival of the TSTAMP_REPLY, thus introducing an extra variable delay.

That this extra delay can be a significant source of error is evident from the following. Mills in [MILL88b] says that "Timestamps are determined by copying the current value of the logical clock to a timestamp variable when some significant event, such as the arrival of a message, occurs. In order to maintain the highest accuracy, it is important that this be

done as close to the hardware or software driver associated with the event as possible". Also Danzig [DANZ91], when discussing influences on the delay between a message being received from a network and being delivered to a user process, says that "the largest randomizing influence on a computer with a heavy workload is the latency to schedule the user process for which a message is destined".

A potential problem with recording the t_3 timestamp in the ICMP layer is the question of where to store it. As noted in part (2) of Subsection 8.2.5, the TSTAMP_REPLY message contains three timestamps: *originate*, *receive* and *transmit*. In SunOS, the *receive* and *transmit* timestamps are set to the same time, so t_3 can be stored in the transmit field of the received TSTAMP_REPLY message, with no loss of information.

The SunOS kernel was modified to record t_1 and t_3 in the ICMP layer, and to store timestamps from the D-M clock, formatted in units of tens of microseconds since midnight UTC modulo four hours, in both TSTAMP and TSTAMP_REPLY messages.

8.3.3 Estimating the Global Time

Probabilistic clock synchronisation algorithms, such as `uoctimed`, are based on the fact that where the round trip time between sending a message and receiving the reply is close to the minimum rtt, then the remote clock estimation error due to variation in message delay must be small [CRIS89], [GUSE89], [VERV91]. After the end of each resynchronisation interval, `uoctimed` running on a slave exchanges ICMP messages with the master in an attempt to get a reply within some threshold rtt. If such a reply is received, then the clock difference estimate is based on the information in this reply, as described in (4) below. To implement this solution, three issues had to be addressed, as described below.

(1) The choice of the threshold rtt

Measurements for the two workstations used for initial testing showed that the minimum rtt for that workstation pair was around 2.88ms. Based on empirical data 3.1ms was chosen to be the threshold rtt, and `uoctimed` version 2.1 had this value compiled into it. (Because the minimum rtt varies with the networking environment, version 2.2 determines the threshold rtt dynamically, rather than statically, as will be described in Section 8.5). By way of comparison, TEMPO has a 20ms threshold rtt compiled into it. The threshold rtt is a parameter of Cristian's algorithm.

(2) The number of attempts made to get a reply within the threshold rtt

With TEMPO, attempts are made until:

(i) a reply has been received with rtt less than the threshold rtt, or

(ii) 5 replies with rtt greater than the threshold rtt have been received, or

(iii) 10 messages have been sent.

We chose the same parameters, and the choices have worked well in practice. Usually, only 1 or 2 messages need be sent. In Cristian's algorithm this number, *k*, is a parameter of the algorithm.

(3) The action to take when no reply is received with rtt less than the threshold

In Cristian's algorithm, a slave leaves the set of synchronised clocks if no replies with rtt less than the threshold rtt are received, but this is a rather drastic solution and one we wanted to avoid. In TEMPO, the differences are calculated from timestamps in the messages with the lowest delays, but large estimation errors can occur. This behaviour was included in `uoctimed` as an option, but the experiments described in Section 8.4 confirmed that large estimation errors do occur when this method is used.

The solution adopted was to compute corrections in these situations from the relative drift rate of the two clocks, that is, the elapsed time since the last correction multiplied by the estimated drift rate. This is reasonable, as the drift rate of two crystal-controlled clocks relative to each other is very close to linear ([ELLI73], [DUDA87], [CARL88], [MILL88b]). In `uoctimed`, if no replies are received with rtt less than the threshold rtt, a slave can correct its clock simply by adjusting for the rate at which it is drifting from the master.

If the -b command line option is specified for `uoctimed`, the lowest message delays in each direction are used for corrections where no replies are received within the threshold rtt, otherwise corrections are based on an estimated drift rate in such situations. An estimated drift rate is supplied initially but, after every hour of operation, `uoctimed` recomputes the drift rate from the total adjustments made to the local clock since the last drift rate calculation. The accuracy achieved with each method is discussed in Section 8.4.

Being able to calculate corrections from an estimate of the drift rate means that one can have a proportion of corrections always based on the drift rate, rather than message passing. The amount of message passing can be reduced, while still achieving good synchronisation [CRIS89]. With the -m option, `uoctimed` allows the user to specify the frequency with which message passing is to be used to determine corrections, with all other corrections being based on the drift rate. For -m 1 message passing is always attempted, for -m 2 message passing is used for every second resynchronisation, for -m 3 message passing is used for every third resynchronisation, and so on. Using estimated drift rate to compute a proportion of corrections has an effect similar to rate adjustment [VERV91].

(4) Computing corrections

For corrections calculated from message passing, the correction is calculated from the three timestamps in the TSTAMP_REPLY message using equation (8.1). Also, -0.5dW (where d is the drift rate of the slave relative to the master, and W is the length of the resynchronisation interval) is added to the correction calculated, so that the master and slave clocks are synchronised half way through the next resynchronisation interval rather than at the start of it. This means that the difference between the slave and master clocks goes from -0.5dW to +0.5dW over the course of the next resynchronisation interval, rather than from 0 to dW, resulting in better synchronisation. The adjustment holds where the slave gains relative to the master and vice versa.

A correction calculated from the estimated drift rate is d multiplied by the time since the last resynchronisation. The -0.5dW adjustment is not required as corrections calculated from estimated drift rates maintain the difference established by the last correction calculated from message passing.

8.3.4 Applying Corrections

A new system call `adjtmr` was installed to allow updating of an offset to the current D-M clock time. The functions that read the D-M clock were changed to return the current clock value plus the value of this offset.

When monitoring is in progress, `adjtmr` limits the absolute value of any change it makes to 1ms. A limit is needed to ensure that local timestamps form a monotonically increasing sequence, and to ensure that elapsed times measured with the clock are accurate. Setting a clock back risks possibility of the timestamp of the next event being earlier than the timestamp of the previous event. For this reason many clock synchronisation systems do not set their clocks back, but instead slow the clock down for a period to achieve the same effect. We chose to allow the clock to be set back by up to 1ms for the following reasons:

(1) The sole use of the clock was to timestamp event records.

(2) The `uoctimed` process is not monitored, that is events relating to its execution are not recorded.

(3) A maximum adjustment of 1ms should be more than enough for the synchronisation at regular intervals of crystal controlled clocks.

(4) Most events are recorded synchronously with process system calls. If `uoctimed` changed the time between successive events of this type, two context switches would be required: one to `uoctimed`, and one back to the process being recorded. The minimum

time between switching from the recorded process to `uoctimed` and resuming the recorded process, was found to be in excess of 1ms. Therefore, even if `uoctimed` set the clock back by 1ms the timestamp of the later event would still be greater than the timestamp of the previous event.

(5) Some events, such as the reception of a UDP datagram, are recorded by the kernel interrupt handling code. Such events can occur just before and just after a correction is made, with no time-consuming context switches required. It is possible that if the D-M clock were set back by 1ms then two events might be timestamped in the wrong order. This is not considered to be a problem of any significance because:

(i) the fastest clock can be chosen as the master, meaning that most slaves perform very few negative corrections.

(ii) most corrections are much smaller than 1ms. In the experiments described in Appendix A in which the -b option was not used, of 4452 corrections calculated from message passing only 128 moved the clock back. Of these, 2 were between -150 and -100μs, 10 were between -100 and -50μs, and 116 were between -50 and 0μs.

(iii) Even if two events' timestamps place them in the wrong order this will not be a problem for recording of interaction networks unless the events are part of the same sub-task.

Finally, a future implementation could avoid instantaneous clock changes in either direction by using a logical clock based on Cristian's offset function described earlier.

8.3.5 Synchronisation With a Time Standard

`uoctimed` does not synchronise to an external time standard because, for the purposes of performance monitoring, it is the relative times of events that are important. In Cristian's algorithm each master is synchronised to a high quality source of UTC.

8.3.6 Fault Tolerance

As the performance measurement software is not fault tolerant, `uoctimed` is not highly fault tolerant. `uoctimed` is tolerant to the loss of messages and to high network delays, but not to failures of the master, or of any of the clocks. Cristian describes ways in which master and clock failures can be handled, if necessary.

8.3.7 Error Bounds

To estimate error bounds, one must consider two main factors: the error in a correction calculation, and the drift that occurs between one resynchronisation and the next. For the

method that we are using, the absolute value of the maximum error made in estimating the difference between the local and remote clocks is:

$$E = (d_{both} - 2d_{min})/2 \quad (8.3)$$

where d_{both} is the rtt measured for some send/reply pair, and $2d_{min}$ is the minimum rtt [CRIS89], [VERV91]. The drift error D is the resynchronisation interval multiplied by the drift rate. To reduce the maximum error, D/2 can be added to the correction, as described in part (4) of Subsection 8.3.3. With respect to drift error at least, one would expect the clocks to be synchronised half way through the resynchronisation interval and to differ by D/2 at either end of the interval, with the maximum error then:

$$M = E + abs(D/2) \quad (8.4)$$

Consider now those replies whose rtt is less than our threshold rtt. With a minimum rtt of 2.88ms, threshold rtt 3.1ms, a 5 second resynchronisation interval, and drift rate of 13μs/second, the maximum error is 137.5μs.

The error between master and slave will therefore range between -137.5μs and 137.5μs. If we consider a second slave synchronised to the same master (with the same drift rate), then the maximum error between the clocks of the two slaves is 275μs.

The above analysis only holds for corrections where a reply was received within the threshold rtt. If the -b option is used there is no bound on the error of corrections based on messages with rtt greater than 3.1ms. Corrections calculated from drift rate should maintain the error of the last correction calculated by message passing, assuming that the drift rate is known reasonably accurately.

8.4 Experiments

The consistency achieved in practice by version 2.1 of `uoctimed` has been checked in a number of experiments. We now summarise important results, with a detailed description of the experiments given in Appendix A.

In the experiments, two D-M clocks were synchronised, with the clock of the master node measured as gaining 13μs/second relative to the clock of the slave node running `uoctimed`. The corrections made by `uoctimed` during a number of experiments were recorded. With a resynchronisation interval of 5s, the expected mean correction was 65μs/s. It is shown in Appendix A that if any corrections recorded are outside the range -145 to 275μs then the maximum synchronisation error must be greater than the 137.5μs maximum error calculated above.

Experiments were conducted in three groups: conditions of "typical" workload, conditions of high workstation load, and conditions of high network traffic.

8.4.1 Typical Workload

Five experiments were conducted. In some experiments the -b option was specified, and in others it was not. Also, 1, 4, and 12, were tried as values for the -m option (the -b and the -m options are described in Subsection 8.3.3).

Of 7403 corrections calculated from message passing, 93.5% were in the range 0 to 150μs. For experiments where the -b option was not specified, no corrections were observed outside the range -145μs to 275μs. It was concluded that `uoctimed` achieves very good synchronisation.

The drift rate recalculations and corrections computed from the drift rate worked as expected. In one experiment, the value of the -m option was 12, so 11 of every 12 corrections were calculated from drift rate estimates. The correction distribution for this experiment was nearly identical to those of experiments where values of 1 and 4 were used for the -m option. There seems no reason why the intervals between messages could not be substantially increased, to rely mostly on estimated drift rate to make corrections.

In one experiment, a run of 60 correction attempts in succession was observed where the rtt achieved was greater than the threshold rtt. The cause, discussed further in Section 8.5, was that the true minimum rtt had increased for that period.

8.4.2 High Machine Workload

During the experiments described in Subsection 8.4.1, it was found that the load on both the workstations and the network was fairly light. To assess how `uoctimed` performed under conditions of heavy load, artificial loads were placed on both the master and slave nodes using a benchmarking suite.

Even under conditions of very heavy workload, `uoctimed` maintained good synchronisation. Because of the heavy workload many of the resynchronisation periods were significantly longer than 5 seconds, causing the correction distribution to be somewhat skewed. In these experiments, 80.6% of corrections were between 0 and 150μs, and 96.6% of corrections were between 0 and 250μs.

Also, in one of these experiments the -b option was used (this option instructs `uoctimed` to calculate corrections from messages every time message passing is performed, even when the lowest rtt of any reply is greater than the threshold rtt). This experiment highlighted the dangers of using the -b option. In the most extreme case an erroneous correction of 4710μs was made based on a message with an rtt of 12.56ms. Similar examples were also observed in the experiments described in Subsection 8.4.1,

such as where a 500μs correction was made based on a message with an rtt of 4.57ms. Use of the -b option was therefore abandoned.

8.4.3 High Network Traffic

During the experiments described above, traffic on the ethernet was almost always light. To assess how `uoctimed` performed under heavy network load, a Sun utility was used to produce artificial network load.

In two experiments with high network traffic, workload on the master and slave was the typical workload. In each of these cases, correction distributions were very similar to those of the experiments described in Subsection 8.4.1.

In a final experiment, both the network traffic and the master and slave workload were high. Even in these extreme conditions `uoctimed` achieved good synchronisation, with 74% of corrections in the range 0 to 150μs, and all corrections in the range consistent with a maximum error of 137.5μs. The workload was so heavy that only 30% of the cases in which message passing was attempted resulted in replies within the threshold rtt.

The results of both sets of follow-up experiments confirmed the conclusion reached for the experiments in Subsection 8.4.1. Despite the heavy workload, nearly all corrections based on message passing were close to the correction values expected. Understandably, there were under heavy load more attempts at synchronisation by message passing for which the message rtt was greater than the threshold rtt. Calculating corrections from such messages would have resulted in substantial errors.

8.5 Version 2.2

A major problem with version 2.1 of `uoctimed` is that the assumption that minimum rtt is constant does not hold (note that Cristian does assume a constant minimum rtt [CRIS89]). This problem, found when the period of 60 long round trip times described in Subsection 8.4.1 was investigated, occurs because a Sun 3/50 with a non-blank screen is slower (by more than 20% [DANZ90]) than a Sun 3/50 with a blank screen. Further investigation showed that minimum rtt varied from 2.68ms to 3.25 ms, depending on the state of the screen of each workstation. The fixed threshold rtt of 3.1ms is inappropriate in the case where both screens are blank (the maximum error increases by around 100μs) and in the case where both screens are non-blank (all rtt will be above the 3.1ms threshold). Simple solutions to this problem include:

1. Disabling the `screenblank` program on both machines.

2. Raising the threshold rtt to around 3.45 ms.

Neither solution is satisfactory. It was decided to make the determination of the minimum rtt, and therefore the threshold rtt, dynamic rather than static. This also enables `uoctimed` to operate where the threshold rtt is quite different from the 3.1ms compiled into version 2.1. For selected pairs of machines on the campus LAN, minimum rtt of between 1.68ms and 7.39ms were recorded. Modifications to `uoctimed` which enable dynamic determination of minimum rtt are described in Subsection 8.5.1.

Two other changes were made for version 2.2. First, the maximum error allowable between the slave's clock and the master's clock is specified as a command line argument in version 2.2 (Subsection 8.5.2). Second, drift rates and minimum rtt are recorded in what is known as a drift file (Subsection 8.5.3).

8.5.1 Determining Minimum Rtt Dynamically

For an algorithm that is to dynamically re-evaluate the current minimum rtt, the main design questions are: when to change the current minimum rtt, and the amount to change it by. Our assumption was that true minimum rtt is constant for relatively long periods (tens of minutes), and changes nearly instantaneously from one value to another. This model applies to the behaviour of the minimum rtt observed for the Sun 3/50s, where changes occurred when a screen changed from being blank to non-blank or vice versa, and the true minimum rtt between such events was very stable.

If a reply is received with rtt less than the current minimum, then clearly the current minimum is wrong, and should be decreased immediately to the smaller rtt. The question of when to increase the minimum rtt is more difficult, as it is expected that most rtt will be greater than the current minimum. Two mechanisms for increasing minimum rtt have been implemented, one for each of the following situations.

1. The true minimum rtt increases to a value above the current threshold rtt. The increase becomes apparent straight away as all subsequent messages have rtt above the threshold. Version 2.2 handles this type of increase in the following way. If 20 messages in succession are observed with rtt above the threshold rtt, then the minimum rtt is increased to the smallest of the 20 rtt.

2. The true minimum rtt increases to a value below the current threshold rtt. The increase is not so obvious. The number of TSTAMP messages sent per correction is likely to rise, as are the number of corrections where no reply is received whose rtt is less then the threshold rtt. This type of increase results in more accuracy than the user requires (see Subsection 8.5.2), so overhead is greater than that necessary to sustain the required accuracy. In an extreme case (as was observed on one occasion) the new true minimum rtt is just below the current threshold rtt. In this situation most message passing corrections

fail to get a reply below the threshold rtt in their 5 attempts, but just enough replies are received within the threshold rtt to prevent mechanism one from increasing the minimum rtt. The solution adopted is that if the current value of the minimum rtt is not observed for 60 rtt in succession, then the minimum rtt is set to the smallest rtt of the 60.

A danger with both of these methods is that false positives might occur, that is the minimum rtt might be increased when the true minimum rtt has not changed. Then for a period (until replies with smaller rtt are received) accuracy will be decreased, as replies will be accepted with longer rtt than should be accepted. Based on experimental data collected with version 2.1 methods one and two will result in very few false positives. For method 1 to produce a false positive there must be 20 successive rtt greater than the threshold rtt, with no change in the true minimum rtt. This situation is not recorded as having happened in any of the experiments, even in an experiment where the load on each Sun and on the ethernet was very high.

The second increase method does give rise to some small oscillations in the recorded value of the minimum rtt. Version 2.2 was synchronising clocks in two Sun 3/50s, with screens on both machines blank. The true minimum rtt appears to be 2.67ms, but this is observed only occasionally, with often more than 60 rtt between two of 2.67ms. The second increase method did increase the minimum rtt to 2.68 and 2.69, but such small fluctuations have little affect on the maximum error.

If no information is available from the drift file (see Subsection 8.5.3) the minimum rtt is initially set to 20ms from which it reduces to a value close to the true minimum rtt within a few corrections.

8.5.2 User Specified Maximum Errors

In version 2.1, the user had no control over the maximum error to be allowed between the clocks of the slave and the master. In version 2.2, the maximum error (e) is specified on the command line, and `uoctimed` calculates the threshold rtt (t) as:

$$t = m + 2e - dr \tag{8.5}$$

where d is the drift rate of the master relative to the slave, r is the resynchronisation interval, and m is the current minimum rtt.

The threshold rtt must be chosen to give some leeway over the minimum rtt, so if t as calculated above is less than m + .05ms then t is set to m + .05ms, and a warning message is printed.

As m and d are determined dynamically, t is recalculated each time that the value of either m or d changes.

8.5.3 Drift File

Each time the drift relative to the master or the minimum rtt is recalculated, both are written to a drift file. The drift file name contains the host name of the master, so if a slave synchronises to different masters at different times the drift rate and minimum rtt applicable to each master will be stored separately. When `uoctimed` begins, it attempts to get initial values for drift and minimum rtt from the appropriate drift file. If this attempt is unsuccessful (usually because the master hasn't been used by the slave before) then defaults of 0 (for the drift rate) and 20ms (for the initial minimum rtt) will be used. The correct drift rate is calculated after 1 hour, and the minimum rtt will quickly approach the true minimum rtt.

8.6 Summary

The Sun 3 system clock resolution of 20ms is inadequate for our purposes. D-M clocks providing microsecond resolution were installed.

None of the clock synchronisation algorithms available (NTP and TEMPO) provided clock synchronisation of sufficient accuracy, so `uoctimed` was developed. `uoctimed` uses a probabilistic clock synchronisation algorithm and an adaptation of the remote clock estimation techniques of TEMPO. In experiments, `uoctimed` was shown to provide very good synchronisation, with all corrections recorded consistent with a maximum error of 137.5μs, acceptably close to our target of 100μs.

`uoctimed` incorporates a number of features that we think are novel, with the first two being worthwhile enhancements to Cristian's algorithm [CRIS89]:

(1) Estimated drift rates are used to calculate corrections whenever a message round trip time is greater than some threshold round trip time. Much better synchronisation was achieved than by using times from messages with long round trip times, the method used by TEMPO. Using drift rates is also preferable to Cristian's solution, where a slave must leave the group of synchronised clocks if after k attempts it has failed to get a reply from the master within the threshold round trip time.

(2) High accuracy requires dynamic determination of the minimum round trip time. `uoctimed` requires the threshold round trip time to be just above the minimum round trip time, but this minimum can change over time. Cristian's algorithm assumes a constant minimum round trip time.

(3) The maximum error bound can usefully be specified as a parameter, allowing a trade-off between overhead and accuracy.

(4) Our experiments showed that a large proportion of corrections can be calculated from the drift rate alone, greatly reducing the message passing overhead.

`uoctimed` achieves good synchronisation, although some other high precision clock synchronisation algorithm could be substituted for it for the purpose of recording interaction networks. The correction approach outlined in Subsection 8.2.2 an interesting possible substitute. `uoctimed` provides a global timebase accurate enough to support the INMON interaction network monitor, which is the subject of Chapters 9, 10, and 11.

Chapter 9

INMON: Probes and the Event Recorder

In this chapter, the probes and event recorder of INMON are described in detail. To allow identification of tasks, and the construction of an interaction network for each task, the following must be done:

(1) Probe points must be selected. The 53 INMON probe points are described in Section 9.2.

(2) Space must be made available so that a sub-task number can be associated with each process and with each message (Subsections 9.3.1 and 9.3.2).

(3) Probe parameters must be selected (Subsection 9.3.3).

An event recorder on each node is responsible for storing events in a log file, as described in Sections 9.4 and 9.5. The analysis tools discussed in the next chapter can be used to construct interaction networks from events recorded over many log files from many nodes.

Decisions on the probes to install, and the design of the event recorder, were both heavily influenced by the nature of the SunOS operating system. An overview of SunOS is therefore given in Section 9.1 before the probes and the event recorder are discussed. Note that in the remainder of this thesis, the term "SunOS" refers to version 4.0 of SunOS.

This chapter describes all of the components of INMON that have been implemented in the SunOS kernel. The kernel changes made to support clock synchronisation, discussed in Chapter 8, are not considered to be part of INMON. The only INMON components discussed in this chapter that are executed in user mode are parts of the event recorder, described in part (1) of Subsection 9.5.2.

Unix manual pages for INMON programs and system calls appear in Appendix B.

9.1 SunOS Overview

A brief overview of SunOS was given in Subsection 7.5.1. Using the operating system classifications introduced in Chapter 2, we can describe SunOS as a process-oriented, network operating system. Process management in SunOS is described in Subsection 9.1.1, and interprocess communication is covered in Subsection 9.1.2. User

input is discussed in Subsection 9.1.3. Finally, a brief overview is given in Subsection 9.1.4 of the probes chosen to be installed for INMON.

9.1.1 Process Management

Processes in SunOS conform to the process definition given in Subsection 2.2.1, that is a process is an address space (cluster), in which a single thread of control executes. SunOS provides a lightweight process library that supports multiple threads within an address space. The library is, however, little used so we have chosen to ignore it.

A new process is created using the `fork` system call. The new (child) process is a copy of the creating (parent) process Processes form a hierarchy, with the parent of a Process P_A being the process that created (forked off) Process P_A. If the parent of Process P_A terminates then, to preserve the hierarchy, process 1 (`init`) becomes the parent of Process P_A. A parent process can wait for a child to terminate by using the `wait` system call. A process can switch execution from the program that it is running to a new program using the `exec` system call. A process terminates normally by making an `exit` system call.

9.1.2 Communication

In early versions of Unix, the *pipe* was the only mechanism provided for interprocess communication [RITC74]. A pipe is a stream-oriented, reliable, unidirectional communication channel. It is designed for communication between two processes on the same node.

A major design goal for Berkeley Unix was that it should provide support for the Darpa Internet communication protocols [QUAR85]. To provide this support, the designers introduced the *socket* abstraction. A socket is a communication endpoint. System calls are provided for naming sockets, for connecting sockets, and for passing data between two sockets. Communication between two sockets is full duplex, whereas a pipe provides unidirectional communication.

The socket software is constructed so that it can provide access to many different families of communication protocols, known as *domains*. In SunOS, two domains are supported: the Unix domain, and the Internet domain. The Unix domain provides intra-node communication only and, in Berkeley Unix, pipes are implemented as a pair of communicating sockets in the Unix domain. The Internet domain provides intra-node and inter-node communication, using the Darpa Internet communication protocols (which are most often referred to as TCP/IP).

The two predominant styles of communication supported by sockets are: stream, where communication is between two connected sockets, is reliable, preserves message sequence,

and does not preserve message boundaries; and datagram, where communication is unreliable, may not preserve message sequence, and does preserve message boundaries. Each socket has associated with it two buffers: a send buffer, and a receive buffer. The receive buffer is used to hold data that has been received by a socket, and has yet to be transferred to a process. The send buffer is used to hold data that has been transferred from a process to a socket, and has yet to be sent, or that has been sent but is yet to be acknowledged.

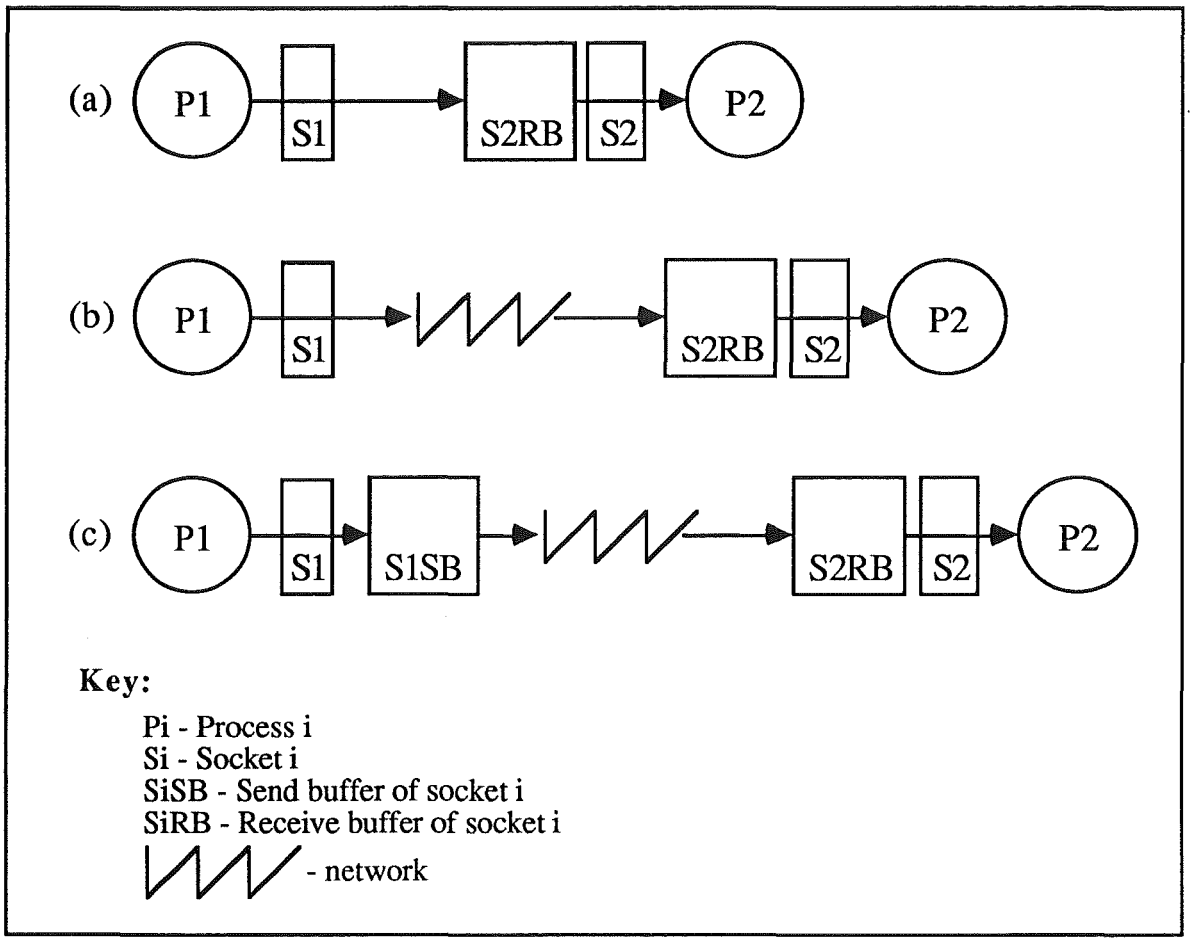


Figure 9-1: Ways in which a message can travel between sockets

The various ways in which a message can travel between two sockets are shown in Figure 9-1. In all cases, Process P1 has associated with it Socket S1, and Process P2 has associated with it Socket S2, and Process P1 is sending a message to Process P2 through Sockets S1 and S2.

Figure 9-1(a) applies to datagram and stream sockets in the Unix domain. Because Sockets S1 and S2 are on the same node, the message is copied directly into the receive buffer of Socket S2. Figure 9-1(b) applies to datagram sockets in the Internet domain.

The message is sent directly onto the network and, when the message is received, it is copied into the receive buffer of Socket S2.

Figure 9-1(c) applies to stream sockets in the Internet domain. First, the message is copied into the send buffer of Socket S1. Then, it is sent onto the network, but a copy remains in the send buffer of Socket S1. Finally, the message is received, and it is copied into the receive buffer of Socket S2. When data is successfully received at Socket S2, an acknowledgement is sent, and the acknowledged data is removed from the send buffer of Socket S1 (the acknowledgement is not shown in Figure 9-1(c)). Note that send buffers are used only for Internet domain stream sockets.

Several additional interprocess communication mechanisms were introduced by AT&T in Unix System V [BACH86] These mechanisms enabled: intra-node message passing, processes to share areas of their address spaces, and processes to synchronise access to shared memory by using semaphores.

Signals, present in all versions of Unix, are software interrupts sent to a process to notify it of the occurrence of some event, with most signals sent as a result of some sort of error. Most signals terminate a process by default, although nearly all signals can be ignored, or can cause the execution of a user-specified function. Processes can send signals to each other, so signals provide a crude form of interprocess communication.

As the structure of SunOS 4.0 is heavily influenced by Berkeley Unix, sockets are the primary interprocess communication mechanism, although the System V interprocess communication mechanisms described above are also provided.

9.1.3 User Input

Our Sun workstations have four sources of direct user input: the workstation keyboard, the workstation mouse, and two serial ports. The device drivers for all of these devices communicate with the remainder of the kernel using the streams mechanism [SUN88d], a System V feature that has been incorporated into SunOS. An example of a stream is shown in Figure 9-2.

In Unix, user processes access all devices, including the user input devices, through the file system. Each device has one or more names in the file system, usually in the `/dev` directory. A user process accesses a device by opening one of the device's file names, and then reading and/or writing to the open file. When a device file is opened, the kernel detects that a device is being accessed, rather than an ordinary disc file.

Where the device has a streams interface, the device-independent part of the kernel communicates with the device by reading input from, and writing output to, the stream head. Information passes through the zero or more modules between the stream head and

the driver, with the driver performing device I/O. Modules can perform functions such as buffering and multiplexing. For example, a module in a stream associated with a terminal device might, among other things, buffer input characters, so that only complete lines of input are transferred upstream. Data flows within a stream can be regarded as messages.

User inputs are detected by device drivers, and flow upstream to the stream head.

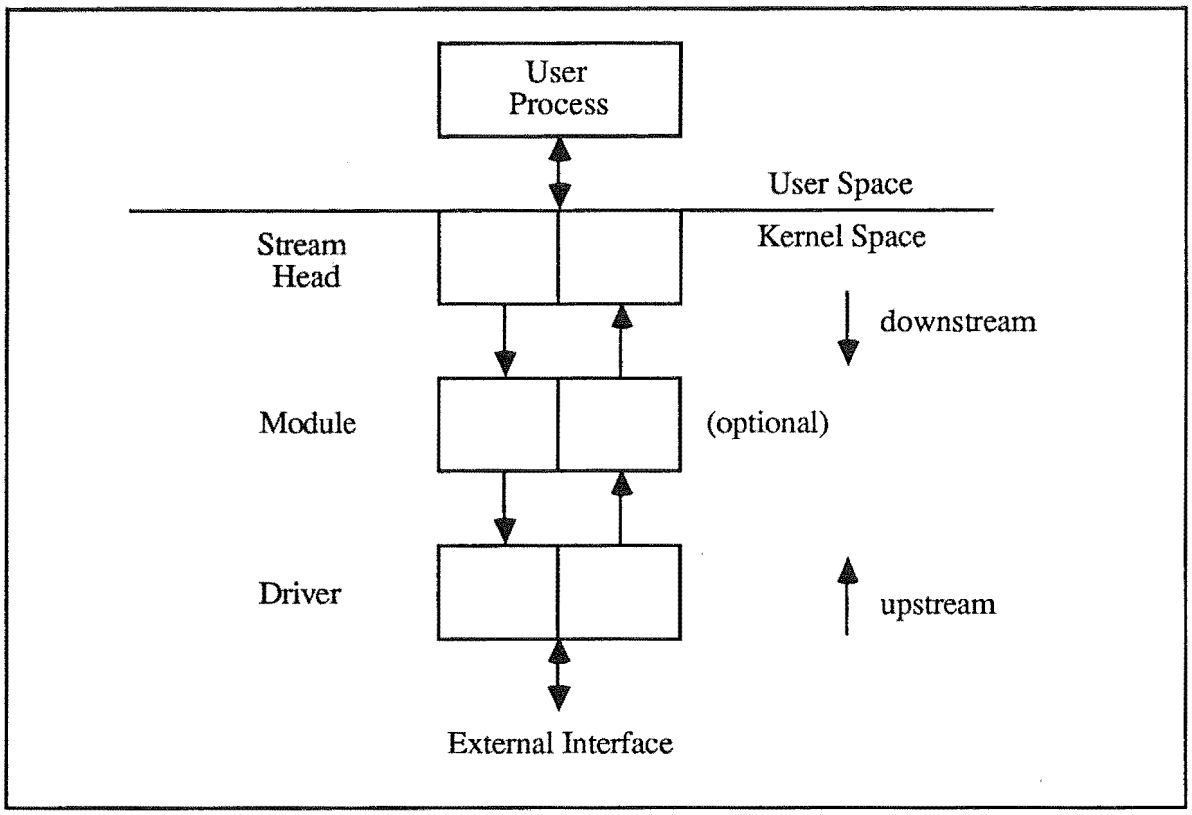


Figure 9-2: Components of a stream (taken from [SUN88d]).

9.1.4 Summary

Interaction networks must include process management events, communication events, and user input events. Also, interaction networks should include events relating to the use of various objects of interest. As part of INMON, the following probes, described in detail in Sections 9.2 and 9.3, have been installed:

(1) For process management: Probes to record process creation and termination, and probes to record events related to a parent process waiting for a child process to exit.

(2) For communication: Probes to record communication events for datagram and stream sockets in the Unix domain, and datagram sockets in the Internet domain. The

System V interprocess communication mechanisms have not been instrumented, as they are little used in SunOS 4.0.

(3) For user input: Probes to record all inputs from directly connected devices.

(4) For object-use: Probes to record use of system-level objects, such as the processor, discs, and some system queues.

9.2 Probes Installed

Decisions on what probes to install were heavily influenced by the main aim in the construction of INMON: to demonstrate that the events that make up each task can be identified, and assembled into an interaction network. The probes that record sub-task start and sub-task end events are described in three groups:

(1) probes that record user input events in (Subsection 9.2.1),

(2) probes that record process management events in (Subsection 9.2.2), and

(3) probes that record communication events in (Subsection 9.2.3).

The remaining probes, described in Subsection 9.2.4, record events that show use of some system resources, in particular use of the processor and disc. Finally, in Subsection 9.2.5, the states are defined that a sub-task can take based on the object-use events recorded by INMON.

In INMON, each of the 53 probes, and the event record that it produces, is identified by a name. These names are listed in Table 9-1.

All probe points are in the kernel. Deciding on the location of each probe has required a detailed understanding of the operation of the relevant part of the kernel. Large parts of the SunOS 4.0 kernel are little changed from the 4.2 BSD kernel, and the kernel also contains some code introduced from System V Unix, such as code for streams and for the System V interprocess communication mechanisms. Descriptions of the internal operation of the BSD kernel and the System V kernel can be found in [LEFF88] and [BACH86] respectively.

9.2.1 User Input

Each user input is detected by a device driver, which then puts the input into a stream. Probes to detect user inputs are best placed within device drivers but, in INMON, these probes have been placed in the device-independent kernel functions that retrieve data from a stream head. This placement has some drawbacks. Any time that a lexeme or group of lexemes spends within the stream, including the time spent waiting at the stream head, is

Subsection	Probe names
9.2.1	CONS_INPUT_BEGIN, CONS_INPUT_END, KBD_INPUT_BEGIN, KBD_INPUT_END, MS_INPUT_BEGIN, MS_INPUT_END, TERM_INPUT_BEGIN, TERM_INPUT_END
9.2.2	EXEC, EXEC_ERROR, EXIT, FORK, SIGPAUSE_END, SIGPAUSE_START, WAIT_SLEEP, ZOMB_EXIT
9.2.3	ACCEPT_JOIN, ACCEPT_START, ACCEPT_WAIT, BKFLOW_RCV, BKFLOW_SEND, PACKET_DROP, RPC_CALL, RPCCLNT_RECV, RPCCLNT_SEND, RPCCLNT_WAIT, RPCSVC_RECV, RPCSVC_SEND, RPCSVC_WAIT, SOCK_CANTRCV, SOCK_CLOSE, SOCK_CONNEND, SOCK_CONNFORK, SOCK_EPIPE, SOCK_RCV, SOCK_RCVWAIT, SOCK_SBREL, SOCK_SEND, SOCK_SENDWAIT, SOCK_SHTDN, SOCK_STREAM_RCV, UDP_RCV, UDP_SEND
9.2.4	BIOWAIT_START, BIOWAIT_END, RESUME, SD_STRATEGY, SETRQ, SWAPIN_BEGIN, SWAPIN_END, SWAPOUT_BEGIN, SWAPOUT_END, SWTCH

Table 9-1: Probe names grouped by subsection

not accounted for in interaction networks recorded by INMON. Also, if a stream (say) assembles input characters into lines, the input of a line is seen as a single input rather than as many individual inputs.

The reason for not placing the probes in the device drivers is that if the probes had been in the device drivers, code would have been required to pass the sub-task number generated for each input from the device driver, through all of the stream modules, to the stream head. Such code could certainly have been written, as shown by the experiences with the communication probes described in (3), but it was felt that the effort required was not justified.

Eight probes, in four pairs, have been installed to record user input events. One probe in each pair records the event of a process starting to wait for input, and the other records that the process has received input. Each "input received" event is the source event of a new task. Pairs of probes have been installed to record inputs from the console device, the keyboard device, the mouse, and the serial ports. User input from the workstation keyboard is directed to either the `console` device driver or the `keyboard` device driver. Put simply, keyboard input goes to the `keyboard` device driver when a graphical user interface is in use, and the `console` device driver otherwise. One pair of probes is used to record inputs from both serial lines, with the number of the serial line stored as an event-dependent parameter.

The eight probe names, in <begin-input-wait, input-received> pairs, are:

(i) console <CONS_INPUT_BEGIN, CONS_INPUT_END>

(ii) keyboard <KBD_INPUT_BEGIN, KBD_INPUT_END>

(iii) mouse <MS_INPUT_BEGIN, MS_INPUT_END>

(iv) serial line <TERM_INPUT_BEGIN, TERM_INPUT_END>.

Many terminal sessions on our Sun workstations are remote, so user input arrives in network messages rather than from directly connected devices. In the example given in Subsection 4.2.5, user input arrives in this fashion. For a remote session, the user input should be recorded when input, and tracked across the network. INMON does not currently provide this tracking: to do so would require the instrumentation of TCP/IP (see section (3) below) and streams. No particular problems are foreseen in adding this instrumentation, however.

9.2.2 Process Management

Process management probes are now described in five groups.

(1) Process creation

A new process is created when an existing process makes a `fork` system call. The FORK probe is executed whenever a new process is created.

(2) Program execution

A process requests that it begin execution of a new program through an `exec` system call. The EXEC probe is executed whenever an `exec` system call is made. If an error occurs, and the new program cannot be executed, the EXEC_ERROR probe is executed before `exec` returns.

(3) Process termination

Normal process termination occurs when a process makes an `exit` system call. Most signals can also terminate a process. When a process terminates, its execution ceases and all of its resources, except its `proc` structure, are released. The `proc` structure of a process is a kernel data structure that holds many pieces of process state, and is kept permanently in memory. On process termination, several pieces of information are saved in the `proc` structure, and the process enters the zombie state. The recorded information includes the cause of the process termination, and some statistics on resource usage.

Process P_A remains in zombie state until its parent performs a `wait` system call, and Process P_A is the zombie process selected by `wait`. At this point `wait` returns the information recorded in the zombie's `proc` structure to the parent, and then releases the

`proc` structure thereby terminating the zombie process. The `init` process calls `wait` whenever it has a zombie child process.

The EXIT probe is executed when a process terminates and enters zombie state. The probe ZOMB_EXIT is executed when a process collects information from a zombie child process, and lays the child finally to rest. Probes that record events involved in a parent waiting for a child to exit are discussed in (4).

(4) Waiting for a child to exit

A process can wait in two ways for one of its child processes to terminate. Firstly, a process can use the `wait` system call. If the process has no children then `wait` returns an error. If one or more zombie children already exist, then one is selected and `wait` returns with information on the selected zombie child. If no zombie children exist, the parent process blocks until a child terminates, at which time `wait` returns information on the terminated child. The WAIT_SLEEP probe is executed whenever a process blocks within `wait`. Note that there are non-blocking variants of `wait`, where `wait` returns immediately if no zombie children exist.

The second way in which a process can wait for a child process, is for it to wait for the signal that is sent to a parent process whenever a child process terminates (by default this signal is ignored by the parent). The parent process makes a system call that suspends it until a signal arrives, the `sigpause` system call in the case of SunOS. When a "death of a child" signal (or any other) is received, the parent is resumed. The probe SIGPAUSE_START is executed when a process is about to be suspended by the `sigpause` system call, and the probe SIGPAUSE_END is executed when a process is resumed. These two probes record events relating to any `sigpause` call, but are intended primarily for the case where a parent is waiting for a child to terminate

(5) Finishing work for a sub-task

A process may show the end of work for a sub-task by terminating. The EXIT probe described in (3) records these events. In all other cases, the fact that a process has finished work for one sub-task is implied by the occurrence of an event which associates the process with another sub-task, such as the event of a process receiving a message.

9.2.3 Communication

Probes have been installed to record events that occur in the following styles of socket-based communication:

(i) Datagram style in the Unix domain.

(ii) Stream style in the Unix domain (including pipes).

(iii) Datagram style in the Internet domain (supported by the UDP/IP protocol).

Probes have not been installed to record events relating to stream communication in the Internet domain, which is supported by the TCP/IP protocol. Adding probes to record TCP/IP related events should be possible, as a stream-oriented protocol has been instrumented (the one in the Unix domain), and an Internet protocol has been instrumented (UDP/IP). This issue is discussed further in (3) below.

Discussion of probes installed to record communication events is divided into four sections:

(1) Probes added to record general socket events.

(2) Probes added to record datagram socket events.

(3) Probes added to record stream socket events.

(4) Probes added to record events for kernel remote procedure calls

(1) General socket probes

These probes record events at the highest layer of the socket abstraction. Communication using sockets can be divided into three phases: setup, data transfer, and shutdown. The nature of the setup phase is heavily dependent on the communication style, with probes recording events of this type described in (2) and (3) below. In particular, establishing a connection between two stream sockets is discussed in (3). The probes described in this subsection record some of the events which occur during the data transfer and shutdown phases.

The probes which record successful data transfer are the SOCK_SEND and SOCK_RCV probes. The SOCK_SEND probe is executed whenever a sending process successfully transfers a message to a socket. The SOCK_RCV probe is executed whenever a message is successfully transferred from a socket to a receiving process. If a process performs a blocking read from a socket, and the socket's receive buffer is empty, then the process must wait for data to arrive. The SOCK_RCVWAIT probe is executed when a process is suspended, awaiting the arrival of data.

A process may also be suspended when it attempts to send data. This situation arises most often because of flow control techniques used with stream sockets. When a socket's send buffer becomes full, no more data can be accepted for sending until some of the data in the send buffer has been transferred. The SOCK_SENDWAIT probe is executed when a process is suspended, waiting for space to become available in a socket's send buffer. The BKFLOW_SEND probe is executed when a socket sends notification that it is ready to receive more data. The BKFLOW_RCV probe is executed when a socket receives

notification that the connected socket is ready to receive more data. The reception of the notification may allow a process that is blocked waiting to send to continue.

A process can fully release a socket using the `close` system call. The SOCK_CLOSE probe is executed when the last process with a socket open closes that socket. Using the `shutdown` system call, a process can specify that data transfer to and/or from the socket be disabled. The SOCK_SHTDN probe is executed on every call to `shutdown`. A process attempting to read from or write to a socket may find that it is unable to do so because of a `close` or a `shutdown` performed on the socket itself, or, for stream communication, on the connected socket. The SOCK_CANTRCV probe is executed when a process cannot read from a socket for this reason, and the SOCK_EPIPE probe is executed when a process cannot write to a socket for this reason.

(2) Datagram probes

Datagram communication requires few probes because of its simplicity. A datagram socket can be given a name by the `bind` system call. A datagram socket can be used to send messages to any datagram socket, in the same domain, that has been given a name. A process specifies a datagram destination every time that it sends a message. Because datagram communication is connectionless, that is no logical connection need be set up before two datagram sockets can exchange messages, no probes are needed to record setup phase events for datagram communication.

In datagram style communication, each message is sent as a unit on a "best effort" basis. No guarantees are given that a message arrives at its destination, or that the order in which a sender sends a group of datagrams is the same as the order in which the receiver receives them. If a datagram is received by a socket, and the receive buffer of the socket is full, then the datagram is simply discarded. The PACKET_DROP probe is executed whenever a datagram is discarded.

In the Unix domain, a datagram is copied directly into the receive buffer of the receiving socket. No additional probes are required, as a SOCK_SEND event is recorded when the datagram is put into the receive buffer, and a SOCK_RCV event is recorded when the datagram is removed from the receive buffer.

Datagram communication in the Internet domain is provided by the UDP/IP protocol [POST80]. Most datagrams in the Internet domain are sent across a communication network. Two probes are needed to detect the time a datagram spends travelling between the UDP layers of the communicating nodes. The UDP_SEND probe is executed whenever a UDP datagram is sent, and the UDP_RCV probe is executed whenever a UDP datagram is received. A received UDP datagram is immediately placed in the receive buffer

of the destination socket, and therefore no additional probe is required to record the event of the UDP datagram being added to the receive buffer of the destination socket.

(3) Stream probes

Before a stream socket can be used to exchange data, it must be connected to another stream socket. Full duplex data exchange between the two connected sockets can then proceed. Delivery of data is reliable, in that messages are not lost or reordered, and message boundaries are not preserved. Because message boundaries are not preserved, data that was sent in two different messages may be received as a single message. The non-preservation of message boundaries makes the management of sub-task numbers more complex, as will be discussed in Section 9.3.

The connection process is asymmetric, with one process taking a server role, and the other a client role. The server process creates a stream socket, gives it a name using the `bind` system call, notifies the kernel that it is willing to accept connections on the socket using the `listen` system call, then waits for a connection request using the `accept` system call. The client process creates a stream socket, then requests that it be connected to the server socket using the `connect` system call. The client passes as a parameter to `connect` the name of the socket that it wishes to connect to.

When the connection is established, a new socket is created for the server, and is connected to the client socket. The `accept` call returns the identity of the newly created socket. The client socket and the server socket returned by `accept` are now connected, and communication using the sockets can begin. The server can accept more connections on the original socket.

The ACCEPT_START probe is executed whenever a call to `accept` is made. If no connections are pending, and the `accept` call is a blocking one, then the calling process must block. The ACCEPT_WAIT probe is executed whenever a process blocks in `accept`, awaiting an incoming connection. The ACCEPT_JOIN probe is executed when an incoming connection has been accepted. The SOCK_CONNFORK probe is executed whenever a call is made to `connect`. The SOCK_CONNEND probe is executed when the connection has been established.

The non-preservation of message boundaries in stream style communication requires two additional probes. The SOCK_STREAM_RCV probe is executed whenever a process receives data from a stream socket, and it records details of any message joining and/or splitting that occurred (see part (4) of Subsection 4.2.4). This probe is executed immediately before the SOCK_RCV probe that records the reception of the message. The SOCK_SBREL probe is executed whenever data in a stream socket send or receive buffer is discarded, and serves much the same function as the PACKET_DROP probe. Because

stream sockets provide reliable communication, data is only discarded from stream socket buffers where the connection is terminated before all data has been received.

(4) Kernel RPC probes

The SunOS kernel includes code that provides some of the functionality of Sun RPC. In particular, remote procedure calls on top of the UDP/IP protocol are supported. The NFS implementation within the kernel is one of the users of the kernel RPC facilities. When a client node wishes to have a file operation performed on a remote node, it makes one of the set of remote procedure calls defined by the NFS protocol. The kernel RPC implementation uses sockets to pass messages, but at a level below that of all of the probes discussed in (1) and (3) above.

The RPC_CALL probe is executed whenever a client process makes a remote procedure call. The RPCCLNT_SEND probe is executed when a client process sends a request to the server. The client process must then wait for the reply. The RPCCLNT_WAIT probe is executed when the client process begins to wait for a reply from the server. The RPCCLNT_RECV probe is executed when the reply arrives.

The RPCSVC_WAIT probe is executed when a server process begins to wait for a request. The RPCSVC_RECV probe is executed when the request arrives. The RPCSVC_SEND probe is executed when the server process sends a reply.

Note that remote procedure calls made outside the kernel go through the remote procedure call library, which uses the socket system calls to achieve client-server communication. The RPC group of events described in this subsection, therefore, apply only to RPC calls from within the kernel, with events relating to socket communications recorded for RPC calls made outside the kernel.

9.2.4 Resource Use

We now describe groups of probes that record events related to: processor use, disc use, and process swapping.

Three probes record events relating to processor use. The RESUME probe is executed when a process starts a burst of execution on a processor. The SWTCH probe is executed when a process finishes a burst of execution on a processor. The SETRQ probe is executed when a process is put onto the processor queue. There is no probe to record when a process is removed from the processor queue, as removal from the processor queue is implied by a RESUME event.

Three probes record events relating to accesses made to SCSI discs. (All discs on the Suns used for INMON were SCSI discs). The SD_STRATEGY probe is executed

whenever a process requests a SCSI disc access. The BIOWAIT_START probe is executed whenever a process begins to wait for a SCSI disc access to complete. The BIOWAIT_END probe is executed whenever a process finishes waiting for a SCSI disc access to complete. Note that there is currently no probe to detect when a disc request is removed from the head of a disc request queue, and service begins.

Four probes record events that relate to swapping processes in and out. In early versions of Unix, each process address space was either in memory or stored on disc (*swapped*) in its entirety. Part of the scheduling activities for such systems involved deciding on the set of processes to keep in memory, and swapping processes to and from disc as that set changed.

In SunOS, each process has a demand paged virtual address space, so there is no need to swap entire processes to and from disc, and "swapping" is simply used as a form of thrashing control. If a process has been inactive for a long period, or if physical memory is over committed, a process is swapped out. This simply involves marking the process as swapped; the processor is never assigned to a process marked as swapped. None of the pages of a process are transferred to disc at the time a process is marked as being swapped. The demand paging system will over time reassign all of the pages of a swapped process, being sure to write any dirty pages out to disc. The SWAPOUT_BEGIN probe is executed whenever a process swap out begins, and the SWAPOUT_END probe is executed whenever a process swap out ends.

When a process is swapped in, it is simply marked as runnable; the demand paging system looks after faulting referenced pages back into memory. The SWAPIN_BEGIN probe is executed whenever a process swap in begins, and the SWAPIN_END probe is executed whenever a process swap in ends.

9.2.5 Sub-task States

Because the main goal in constructing INMON was to show that interaction networks could be recorded, more emphasis has been placed on installing probes that record structural events than on probes that record object-use. Object-use events for many system-level objects (as described in part (3) of Subsection 6.1.2) are, however, recorded by INMON probes. At any time, each message or thread is using exactly one system-level object, so the state of a sub-task activity is defined by the system-level object in use during that activity.

Determining sub-task states for message and process activities is discussed in (1) and (2) below. Information from probes that detect structural events and from probes that detect object-use events is used to determine sub-task states.

(1) States for message activities

A message, and its associated sub-task, can be in one of two states: `queued` or `network`. A message in `queued` state is buffered within a node, either waiting to be sent across the network, or waiting to be received by a process. A message in `network` state is in transit on a network link.

In INMON, UDP is the only protocol that has been instrumented in which messages are sent across a network. It is therefore easy for the state of a message sub-task to be determined: between a UDP_SEND event and a UDP_RCV event a message sub-task is in `network` state, and at all other times a message sub-task is in `queued` state.

The probes that record sending a UDP message on a network link, and receiving a UDP message from a network link, are located in the UDP modules of the kernel. A `network` state time, therefore, includes the time required by lower protocol layers, as well as the message transmission time itself. Also note that computing a `network` state time involves the comparison of times recorded on different nodes. The accuracy of the clock synchronisation achieved by `uoctimed` means that such comparisons are valid.

(2) States for process activities

A process, and its associated sub-task, can be in one of fourteen states. The three main process states in any system are ready (to execute), running (on a processor), and waiting (for some event to occur, such as for an I/O request to complete) [SILB91]. In INMON, the ready and running states are called `ready` and `run`, and there are nine states which collectively form the waiting state. Each of these nine states represents waiting on some queue(s) for some particular type(s) of event to occur. The waiting states are summarised in Table 9-2. Note that the time spent in `sleep_disc` state includes both time spent waiting to use the disc and the disc access time.

State name	State description
`sleep_terminput`	Process is waiting for user input.
`sleep_child_wait`	Process is waiting for a child process to exit.
`sleep_sock_send`	Process is waiting to write data to a stream socket whose send buffer is currently full.
`sleep_sock_rcv`	Process is waiting to read data from a socket.
`sleep_RPC_clnt`	Process has sent a kernel RPC request, and is waiting for a reply.
`sleep_RPC_svc`	Process is waiting for a kernel RPC request.
`sleep_accept`	Process is waiting in the accept system call for an incoming connection.
`sleep_disc`	Process is waiting for a disc request to complete.
`sleep_other`	Process is waiting for any reason other than those listed above.

Table 9-2: The process waiting states

There are three other process states:

(1) A process in `zombie` state has terminated, but its parent has not yet waited for it (zombie processes are described in Subsection 9.2.2).

(2) A process in `swapped` state is one that has been swapped out of memory (swapping is described in Subsection 9.2.4).

(3) A process is in `created` state for the brief period between when it is created and when it is first placed on the ready-list.

The INMON process states correspond closely to the nine states for Unix processes identified by Bach [BACH86], as shown by the comparison in Table 9-3. Some INMON states correspond to the combination of two or more of Bach's states, and vice versa.

Bach	INMON
1. User running, and 2. Kernel running	`run`
3. Ready to Run In Memory, and 7. Preempted	`ready`
4. Asleep In Memory	nine `sleep_` states
5. Ready to Run, Swapped, and 6. Sleep, Swapped	`swapped`
8. Created	`created`
9. Zombie	`zombie`

Table 9-3: Equivalences between the process states of INMON and the process states defined by Bach

We now consider events that occur within a process that mark transitions into and out of the fourteen process states.

(i) `run`, `ready`. A process enters `ready` state when a SETRQ event occurs. A process goes from `ready` state to `run` state when a RESUME event occurs. A process leaves `run` state when a SWTCH event occurs.

(ii) `sleep_`. In every sleep state a process is waiting in some queue. An event is recorded when a process is put into a queue and an event is recorded when a process leaves the queue. These events are summarised in Table 9-4. For most sleep states, the process leaves the state the next time it is put onto the ready-list. In the case of the `sleep_child_wait` state entered because of a SIGPAUSE_START event, the SIGPAUSE_END event is recorded so that the signal that ended the signal pause can be

determined. A process was in `sleep_child_wait` state only if the signal received was a "death of a child" signal. In the case of the `sleep_disc` state, a process may be woken briefly several times during the wait, so a BIOWAIT_END event is recorded when a disc request finally finishes. If a process is not recorded as being on any of the monitored queues when a SWTCH event occurs, then it enters the `sleep_other` state.

Sleep state	Event recorded on entering	Event recorded on leaving
`sleep_terminput`	TERM_INPUT_BEGIN	SETRQ
	MS_INPUT_BEGIN	SETRQ
	KBD_INPUT_BEGIN	SETRQ
	CONS_INPUT_BEGIN	SETRQ
`sleep_child_wait`	SIGPAUSE_START	SIGPAUSE_END
	WAIT_SLEEP	SETRQ
`sleep_sock_send`	SOCK_SENDWAIT	SETRQ
`sleep_sock_rcv`	SOCK_RCVWAIT	SETRQ
`sleep_RPC_clnt`	RPCCLNT_WAIT	SETRQ
`sleep_RPC_svc`	RPCSVC_WAIT	SETRQ
`sleep_accept`	ACCEPT_WAIT	SETRQ
`sleep_disc`	BIOWAIT_START	BIOWAIT_END
`sleep_other`	SWTCH	SETRQ

Table 9-4: Beginning and ending events for sleep states

(iii) `zombie`. A process enters `zombie` state when an EXIT event occurs, and is finally destroyed when a ZOMB_EXIT event occurs in which the parent receives notification of the demise of the zombie process.

(iv) `swapped`. A process enters `swapped` state when a SWAPOUTBEGIN event occurs and leaves the state when a SWAPINEND event occurs.

(v) `created`. A process enters `created` state when it is created at a FORK event and leaves the state when the next event in the process, a SETRQ event, occurs.

The potential problems in determining the state for some activities (described in part (6) of Subsection 6.1.3) are easily solved for INMON. The state of the first activity in a newly created process is `created`. The state of a process immediately after the process has received a message is always `run`. Likewise, the state of a process immediately after a source event (the four _INPUT_END events) is always `run`.

Some simple events do not cause changes in state, but are recorded for use in later analysis. For example, EXEC event records provide information on program names, RPC_CALL event records provide additional information on remote procedures called, and SD_STRATEGY event records provide additional information on disc accesses.

9.3 Probe Parameters

In Section 9.2, all 53 probes installed in the SunOS kernel were described. In this section, the parameters of these probes are discussed. In Subsections 9.3.1 to 9.3.3, issues related to parameters that record the sub-task structure are covered. In Subsection 9.3.1, the way in which globally unique sub-task numbers are generated is described. Methods by which sub-task numbers are associated with processes and messages are covered in Subsection 9.3.2, with Subsection 9.3.3 containing a description of the sub-task number parameters recorded for the various events. In Subsection 9.3.4, all other parameters are covered.

9.3.1 Generating Sub-task Numbers

For INMON, sub-task numbers are 32 bit integers, with the most significant 8 bits identifying the node the sub-task number was generated on, and the least significant 24 bits being a sequence number. Each node maintains a kernel variable, `ipid`, which is initialised to 0. To generate a sub-task number, the 8 bits identifying the node are combined with the bottom 24 bits of `ipid`, then `ipid` is incremented. An 8 bit node identifier allows up to 256 different nodes to be distinguished, and with a 24 bit sequence number each node can generate over 16 million sub-task numbers before the sequence number counter overflows.

A special sub-task number, PE_INVALIDIPID, is used to identify processes and messages which are currently part of tasks for which we do not want to record interaction networks, such as background system tasks. No events are recorded for processes and messages that have PE_INVALIDIPID as their sub-task number.

9.3.2 Storing Sub-task Numbers

The storage of process sub-task numbers is discussed in (1), with storage for message sub-task numbers discussed in (2) (datagram messages), (3) (stream messages), and (4) (control messages).

(1) Processes

Each process has associated with it a `proc` structure, that contains many items of process state and is stored permanently in main memory. The kernel contains an array of `proc` structures known as the `proc` table, whose size is fixed at kernel configuration time. Ideally, sub-task number would be a field of the `proc` structure, but adding a new field to the `proc` structure would have caused problems for utilities outside the kernel which extract information from the `proc` table.

Instead, a new table was created to hold the process sub-task numbers. This `shadow proc` table contains the same number of entries as the `proc` table. The sub-task number for the process whose `proc` structure is element I of the `proc` table, is stored in element I of the `shadow proc` table.

(2) Datagram messages

Sub-task numbers must be stored both with datagrams within socket buffers, and with UDP/IP datagrams sent across a network. In SunOS, data in socket buffers is stored as a linked list of `mbuf` structures, where each `mbuf` can hold up to 112 bytes of data. A datagram in a socket receive buffer, is stored as a source address `mbuf`, which contains the address of the socket from which the datagram was sent, followed by one or more data `mbufs`. As only 14 of the 112 bytes are used in the source address `mbuf`, the sub-task number of a datagram in a socket receive buffer can be stored in the source address `mbuf`.

As Unix domain datagrams are stored directly into the receive buffer of the destination socket, the source address `mbuf` provides all of the storage needed for sub-task numbers associated with Unix domain datagrams. In the Internet domain, some means must be found of including a sub-task number with a UDP/IP datagram. The solution adopted was to define a new IP header option [POST81a], and to pass the sub-task number in the IP packet header.

(3) Stream messages

Management of sub-task numbers for datagram communication is straightforward, as messages are kept separate in socket receive buffers, and each message has a source address `mbuf` that has room to store its sub-task number. Management of sub-task numbers for stream communication is potentially much more complex, as message boundaries are not preserved in stream socket send and receive buffers. Address `mbufs` are not required for a stream socket because all of the data arrives from the connected socket. In the context of stream sockets, we define as a message the data sent in each system call that writes data to a stream socket.

To record message boundaries and message sub-task numbers for data in stream sockets, a queue of <message length, sub-task number pairs> can be associated with a socket buffer. The queue is stored in one or more `mbufs`, with a pointer to the queue stored in a field of the `sockbuf` structure associated with the socket buffer. The sum over all elements in a queue of the message length fields is always equal to the number of bytes of data in the socket buffer.

Currently only stream style communication in the Unix domain is instrumented. In the Unix domain, when one stream socket sends to another, the sending socket appends data

directly to the receiving socket's receive buffer. So, at present, only a receive buffer can have a queue associated with it. In the Internet domain, stream style communication is supported by the TCP/IP protocol [POST81c]. Data is initially stored in the sending socket's send buffer, and then is sent across the network and stored in the receiving socket's receive buffer. To instrument TCP/IP, queues would also have to be associated with send buffers, and with TCP/IP messages. A queue could be included with a TCP/IP message as either an additional IP header option, the approach described above for UDP/IP, or as an additional TCP header option [POST81c].

(4) Control messages

There are three situations in which it is desirable to include a sub-task number with what we might call "control messages":

(i) For socket shutdown messages. When a socket is partially or fully shutdown, either the SOCK_SHTDN probe or the SOCK_CLOSE probe is executed. The SOCK_EPIPE and SOCK_CANTRCV probes are executed whenever a process tries to write to or read from a socket which has been disabled for that type of operation. A control message is therefore sent from the process that disabled the socket (or its connected socket for stream sockets) to the process that is attempting the read or write. The SOCK_SHTDN and SOCK_CLOSE probes record the sending of a shutdown message, and the SOCK_EPIPE and SOCK_CANTRCV probes record the reception of a shutdown message

(ii) For backflow messages (stream sockets only). A backflow message occurs when a process receiving from a socket removes enough data from the socket receive buffer to allow a process waiting to send to that socket to continue. The backflow message goes from the reading process to the sending process, against the flow of data. The BKFLOW_SEND probe records the sending of a backflow message by the process receiving data, and the BKFLOW_RCV probe records the reception of a backflow message by the process sending data.

(iii) For connect messages (stream sockets only). When a client tries to connect to a server, a control message accompanies the connect request. The SOCK_CONNFORK probe records the sending of a connect message, and the ACCEPT_JOIN probe records the reception of the connect message by the server process.

For all three types of control message, the associated sub-task number is stored in an unused field of the `socket` structure of the socket receiving the control message. Use of this field for all three types of control message does not result in any conflicts, as one type of message occurs in the setup phase, one occurs in the data transfer phase, and one occurs in the shutdown phase. The backflow and connect messages are applicable only to stream communication. Also, in practice, the shutdown message is nearly always related to a

connected pair of stream sockets. One of the pair is closed, with a shutdown message sent to the second informing it that the connection has been broken. When a process attempts a read or write on the second socket, the process receives a control message, informing it that the operation is not possible because of a socket closure. For datagram sockets, the shutdown control message only occurs under very unusual circumstances.

Currently, the sub-task number of a control message is placed directly into the `socket` structure of the receiving socket. If TCP/IP were instrumented, then sub-task numbers would have to accompany shutdown, backflow, and accept messages across the network. Again, the sub-task number could be passed as an IP option or as a TCP option.

Also, for TCP/IP a new control message would be required, that is sent in reply to a connect request. Such a message is not required in Unix domain stream sockets, as the connection is set up, ready for the server to later accept, during the `connect` system call.

9.3.3 Sub-task Number Parameters

Every event recorded has associated with it the number of the sub-task in which it occurred. Parameters for recording this sub-task number are described in (1). For simple events, no manipulation of the sub-task number is required. For the other event types some manipulation of sub-task numbers is required, and additional sub-task number parameters may be recorded. Source events are discussed in (2), fork events in (3), join events in (4), and sink events in (5). Those events which are related to stream sockets and which have as parameters queues of sub-task numbers are covered separately in (6).

Note that there are no multi-way fork event types recorded by INMON, and the only message join/split event type recorded is discussed in (6).

(1) Recording an event's sub-task number

The number of the sub-task in which an event occurred is an event-independent parameter in the event record constructed by the event recorder. Each probe passes as a parameter to the event recorder, a pointer to a `proc` structure. For process-related events, the pointer points to the `proc` structure of the process in which the event occurred. In this case, the sub-task number of the process is extracted from the `shadow proc` table, and used as the value of the sub-task number event-independent parameter.

For message-related events, a nil `proc` structure pointer is passed to the event recorder. For these events, the sub-task number event-independent parameter is set to 0, and the sub-task number(s) of the message are the first of the event-dependent parameters. These message-related events are SOCK_STREAM_RCV, SOCK_SBREL (both discussed further in (6)), PACKET_DROP, UDP_SEND, and UDP_RCV. All other events are

considered to be process-related, for the purposes of recording the sub-task number of an event.

Note that some of the process-related events involve processes sending or receiving messages. For these events, the value stored in the event-independent parameter that contains a sub-task number is the sub-task number of the process, with the message sub-task number stored as an event-dependent parameter as described in (3) and (4) below.

(2) Source events

Usually, when a source event occurs, a new sub-task number is generated and is assigned to the process which receives the input. If, however, the process that receives a user input has the SPERF bit set in its process flags, then its sub-task number is held at PE_INVALIDIPID, as events relating to this process are not to be recorded. See Subsection 9.5.2 for more information on the SPERF flag.

Probes that detect source events (TERM_INPUT_END, CONS_INPUT_END, KBD_INPUT_END, and MS_INPUT_END) are executed after the new sub-task number has been assigned to the process that is to receive the user input. The first event-dependent parameter of all of these probes is the previous value of the process sub-task number.

(3) Fork events

When a fork event occurs, a new sub-task number is generated and assigned to the new message or process. If the sub-task number of the process in which the fork event occurred is PE_INVALIDIPID, then the task of which this sub-task is a part is not being monitored, so the new sub-task number is also PE_INVALIDIPID. The new sub-task number is recorded as an event-dependent parameter in event records of the fork events, which are: FORK, SOCK_CONNFORK, SOCK_SEND, RPCCLNT_SEND, RPCSVC_SEND, BKFLOW_SEND, SOCK_CLOSE, and SOCK_SHTDN.

(4) Join events

Usually, when a join event occurs, the sub-task number of the process in which the event occurs is set to the sub-task number of the joining process or message, based on the assumption that the process will, after the join, perform work on behalf of the task of the joining process or message. If, however, the process has the SPERF bit set in its flags, then its sub-task number is held at PE_INVALIDIPID, as events relating to this process are not to be recorded. See Subsection 9.5.2 for more information on the SPERF flag.

Probes that detect join events are executed after the new sub-task number has been assigned, and have the previous sub-task number of the process as an event-dependent parameter. The join events are ZOMB_EXIT, ACCEPT_JOIN, SOCK_RCV,

RPCSVC_RECV, RPCCLNT_RECV, BKFLOW_RCV, SOCK_EPIPE, and SOCK_CANTRCV.

(5) Sink events

Sink events are not explicitly recorded, as a sub-task is assumed to end at the last event recorded for it, although some sink event records are added during analysis (see Section 10.2).

There are some situations, however, in which the sub-task number of a process is set to PE_INVALIDIPID, to indicate that no events should be recorded for the process:

(i) When the system is booted, the sub-task number of process 0, the process that runs when the system is booted, is set to PE_INVALIDIPID, to ensure that no attempt is made to record events that occur during the system booting sequence. Because process 0 has a sub-task number of PE_INVALIDIPID, all of its successor processes (that is all processes) will also initially have a sub-task number of PE_INVALIDIPID. Once monitoring has been enabled, a process will be assigned a valid sub-task number whenever it gets a user input, and any successors of that process created subsequently will be given valid sub-task numbers at the time of creation.

(ii) When a process exits, its sub-task number is set to PE_INVALIDIPID.

(iii) When event recording begins, the sub-task number of the process that stores event records in a log file is set to PE_INVALIDIPID, to ensure that no events relating to it are recorded.

(6) Stream socket events with queue parameters

As explained in Subsection 9.3.2, a stream socket buffer may have associated with it a queue, whose elements are <sub-task number, length pairs>, which records the sub-task numbers of all of the messages currently in the buffer. In this section, we consider how these queues are manipulated, the events associated with queue manipulation, and the sub-task number parameters of those events. The discussion applies to Unix domain stream sockets; enhancements described in Subsection 9.3.2 would be required to support internet domain stream sockets.

We now describe the sequence of events that occur when data is sent from Socket S_A to Socket S_B. A process makes a system call to write data to Socket S_A. As space permits, the data is appended to the receive buffer of Socket S_B. Often all data can be added with a single append, but sometimes more than one append is required. For every append, a SOCK_SEND event is recorded, with the event record containing the new sub-task number generated for the message (the data appended). The new sub-task number and the number

of bytes appended to the receive buffer of Socket S_B, are added to the end of the queue associated with the receive buffer.

When a process attempts to read data from Socket S_B, the number of bytes returned (N) is the smaller of the number of bytes requested and the number of bytes in the receive buffer. The first N bytes of the receive buffer are returned to the reading process. Because these N bytes might come from many messages, each with a different sub-task number, a new sub-task number is generated to accompany the message consisting of the N bytes.

A SOCK_STREAM_RCV event (a message join/split event) is recorded when the N bytes are removed from the socket receive buffer. SOCK_STREAM_RCV event records have as parameters a list of <sub-task number, length> pairs describing the origin of the N bytes, and the sub-task number generated to accompany the N bytes. All <sub-task number, length> pairs that refer to data that has been completely removed from the receive buffer, are removed from the buffer's queue. The length field of the queue element now at the head of the queue, is reduced if part of the data it refers to was removed from the socket receive buffer.

A SOCK_RCV event is recorded directly after a SOCK_STREAM_RCV, to represent the process receiving the N byte message. The sub-task number associated with the message received in the SOCK_RCV is the new sub-task number generated by the SOCK_STREAM_RCV event.

If the data in a socket receive buffer is discarded, usually because the socket is closed before all the data has been read, a SOCK_SBREL event is recorded, which has as a parameter the queue of sub-task numbers associated with the socket receive buffer.

9.3.4 Other Parameters

Parameters that do not relate to the recording of sub-task numbers are now discussed. The remaining event-independent parameters are the event type (FORK, EXIT, etc), the time at which the event occurred, and the process id (pid) of the process in which the event occurred (-1 if the event is message-related).

In Table 9-5, we summarise the remaining event-dependent parameters. Event-dependent parameters which have in practice been little used have been omitted from the table.

Event	Event-dependent parameters
FORK	1. pid of the child process
EXIT	1. process exit status 2. pid of the parent process
ZOMB_EXIT	1. pid of the zombie child process 2. resource usage summary for the zombie child process
SOCK_SEND	1. message length 2. sequence number (see part (5) of Subsection 10.4.3)
SOCK_STREAM_RCV	1. message length
SOCK_RCV	1. message length
SOCK_SHTDN	1. direction (send and/or receive) of shutdown
RPCCLNT_RECV	1. message length
RPC_CALL	1. program called 2. version of program called 3. procedure called
RESUME	1. pid of the previous process that used the processor
UDP_SEND	1. destination host
SD_STRATEGY	1. number of disc device 2. read/write flag 3. cylinder number to be accessed
BIOWAIT_END	1. number of disc device waited on
EXEC	1. program name
EXEC_ERROR	1. error code
SIGPAUSE_END	1. number of signal received
TERM_INPUT_BEGIN MS_INPUT_BEGIN KBD_INPUT_BEGIN CONS_INPUT_BEGIN	1. number of the input device the process is waiting on
TERM_INPUT_END MS_INPUT_END KBD_INPUT_END CONS_INPUT_END	1. number of the device that input was received from 2. length of input
RPCSVC_RECV	1. length of request message

Table 9-5: Summary of event-dependent parameters

9.4 Probe Implementation

As noted earlier, all probes installed so far are in the kernel. Each probe consists of a call to `pe_sysperf`, a function that is part of the event recorder (see Subsection 9.5.1). Also, some probes require additional code to perform sub-task number manipulation, and/or computation of one or more of the event-dependent parameters.

The `pe_sysperf` function takes a variable number of arguments, of which the first three must always be supplied: a pointer to the `proc` structure of the process in which the event occurred (a nil pointer for message-related events), the event number, and the number (N) of event-dependent parameters that follow. The remaining arguments consist of N <event parameter length, event parameter pointer> pairs.

Probes outside the kernel can record events using the `perf` system call. A pointer to a complete event record must be passed to `perf`, whereas `pe_sysperf` constructs an event record on behalf of the caller.

9.5 The Event Recorder

The INMON event recorder was developed from the SunOS recorder of audit records. Auditing is part of the Sun C2 security package [SUN88c]. The auditing package allows for the selective recording of events whose occurrence might have security ramifications. Audit event records can be recorded in a log file on each Sun.

The INMON event recorder is now described in two parts: first, the process of creating an event record and storing it in the internal buffers of the event recorder is described; second, the transfer of event records from the internal buffers of the event recorder to a log file is discussed. The transfer of event records is performed by a background (*daemon*) process, running the `perfd` program.

9.5.1 Creating and Buffering Event Records

As described in Section 9.4, probes inside the kernel call the function `pe_sysperf` to record an event , and probes outside the kernel use the `perf` system call to record an event. The `pe_sysperf` function constructs an event record in one of the event recorder's buffers, whereas the `perf` system call simply copies the event record passed as an argument into one of the event recorder's buffers. The way in which `pe_sysperf` constructs an event record is described in (1), with management of event record buffers discussed in (2)

(1) Construction of an event record by `pe_sysperf`

First, `pe_sysperf` decides whether to record the event. The event is not recorded if at least one of the following is true:

(i) Event recording is marked as being "OFF".

(ii) The process pointer passed is not nil, and the sub-task number of the process is PE_INVALIDIPID. Probes that record message-related events must ensure that the sub-task number of the message is not PE_INVALIDIPID, before calling `pe_sysperf`.

(iii) Event recording is disabled for the specified event type.

Note that the `perf` system call does not record the event record if event recording is marked as being "OFF", or if the sub-task number of the calling process is PE_INVALIDIPID (it is assumed that the `perf` system call is used only to record events that occur in the calling process).

If the event is to be recorded `pe_sysperf` then acquires a buffer, in which it creates an event record. The format of an event record is as follows: the header; an array containing the parameter lengths, with 1 element for each event-dependent parameter; and the values of the event-dependent parameters. The header contains the event record length, the number of event-dependent parameters, and values of all of the event-independent parameters.

The header is then filled in as follows. The event code and the number of event-dependent parameters are supplied to `pe_sysperf` as arguments. The record length is calculated from the header length, the number of event-dependent parameters, and the length of each event-dependent parameter value. The time is read from the D-M clock, if one is installed, or the system clock otherwise. If the `proc` structure pointer argument of `pe_sysperf` is not nil, then the pid and sub-task numbers are set to those associated with that process, otherwise they are set to -1 and 0 respectively.

The lengths and values of the event-dependent parameters are then added to the event record. Finally the buffer containing the event record is linked into a list of buffers waiting to be written to the log file.

(2) Buffer management for event recording

When monitoring commences, 100 buffers of length 120 bytes, and 5 buffers of length 1000 bytes are allocated. These sizes were chosen to achieve good storage utilisation. As most event records are small, most of the buffers are small to avoid wasted space, but it is desirable that some allowance be made for longer event records. Two buffer free lists are maintained, one for each buffer size.

A request for a buffer, from `pe_sysperf` or `perf`, is handled in the following way. If the space requested is too big to store in a large buffer, then buffer allocation fails. If a small buffer is big enough and one is available, then a small buffer is allocated. Otherwise an attempt is made to allocate a large buffer and if a large buffer is available it is allocated. If no buffers are available buffer allocation fails, and a warning message is printed on the workstation console. If buffer allocation fails, then `pe_sysperf` and `perf` simply return, unable to record the event.

When `pe_sysperf` or `perf` has copied an event record into a buffer, the buffer is linked to the end of a list of buffers that are waiting to be written to the log file. If, after adding the buffer to the list, the list contains at least 10 small buffers or 1 large buffer, the `perfd` process is put onto the processor queue, so that it can transfer the buffers to the log file.

Originally, the `perfd` process was put onto the processor queue whenever a buffer was ready to be written to the log file. This solution, however, led to excessive context switching. To see why, consider the following situation. Process P_A is allocated the processor. A RESUME event record is placed in a buffer in the event recorder, and the `perfd` process is then put onto the processor queue. Because the `perfd` process uses very little CPU time (and therefore has a high scheduling priority), the `perfd` process pre-empts Process P_A soon after. The `perfd` process writes event record(s) to the log file, and then waits for further events to arrive. The processor then resumes execution of Process P_A, resulting in another RESUME event being recorded, and the `perfd` process being put onto the processor queue, thus starting the cycle again. The processor, therefore, continually switches back and forth between the P_A and `perfd` processes. To prevent this, the requirement was introduced that at least 10 small buffers be pending before the `perfd` process was put onto the processor queue.

Failure to allocate a buffer is a serious problem, as the event record that was to be recorded is lost. Loss of an event record means that one interaction network cannot be constructed from the log file information. The performance analyst must know whether events have been lost during a particular recording session, so a message is written to the system console every time an event is dropped because no buffers were available.

Also, a set of counters is maintained, each of which is incremented when some exceptional event occurs. One counter is incremented every time an event is dropped because of buffer unavailability, and another is incremented every time an event is dropped because its record is bigger than a large buffer. The current values of each counter are printed by the program `dumpperfstat`. Also, `perfd` records in each log file the event drop count at the beginning and at the end of the period over which event records in the file were

recorded. In the experiments performed using INMON, no problems were experienced with dropped records.

9.5.2 Transferring Event Records to the Log File

The `perfd` process is responsible for transferring event records from buffers within the event recorder to a log file. In this section, we describe the operation of `perfd`, with emphasis on normal operation, and a little discussion of error handling.

Basically, `perfd` performs the following functions: it turns on event recording, opens a log file, appends event records to the log file, closes the log file, then turns event recording off. In (1) below, the parts of `perfd` executed in user mode are discussed, and in (2) system calls for enabling/disabling logging, and for recording events in the log file, are discussed. The log file may be stored on a local disc, or it may be stored on a remote disc and accessed using NFS.

(1) User mode components of `perfd`

The major activities performed in user mode by `perfd` are listed below.

(i) Parse the command line arguments. Events which are not to be recorded can be specified as command line arguments.

(ii) Turn on event recording using the `perfon` system call. The list of events which are not to be recorded, constructed in (i), is passed as an argument.

(iii) Open the log file, and write a header record to it. The header includes the time at which the log file was opened, the node on which the log file was recorded, and the current value of the counter of dropped records.

(iv) Make a `perfsvc` system call to record a stream of event records in the log file. `perfd` spends nearly all of its time within the `perfsvc` system call.

(v) The `perfsvc` system call is eventually terminated by a signal. If the SIGTERM signal is received, then `perfd` closes the current log file, turns off event recording using the `perfon` system call, and exits. If the SIGUSR1 signal is received, then `perfd` closes the current log file, opens a new log file (and writes a header record to it), then goes back to step (iv).

As part of closing a log file, an event record of type TRAILER is written to the log file. An event-dependent parameter of TRAILER is the current value of the counter of dropped records.

(2) Event recording system calls used by `perfd`

The two most important event recording system calls used by `perfd` are `perfon`, called to enable and disable event recording, and `perfsvc`, called to append event records to a log file.

When used to enable event recording, `perfon`: registers the event types to be recorded and the event types to be ignored; allocates the buffers; turns on the SPERF bit of the process flags of the current process and sets the process sub-task number to PE_INVALIDIPID; and marks event recording as being "ON". Setting the process sub-task number to PE_INVALIDIPID ensures that events that occur as part of the event recording activity are not recorded, and setting the SPERF flag ensures that the process sub-task number of the `perfd` process is held at PE_INVALIDIPID.

When used to disable event recording, `perfon`: marks event recording as "OFF", resets the SPERF flag of the current process, and frees all of the buffers.

The `perfsvc` system call executes a loop, in which it writes all buffered event records to the log file (whose file descriptor is passed as an argument), then waits for more event records. When a signal is received, `perfsvc` breaks out of the loop and returns.

9.6 Summary

The probes installed, together with sub-task number support, have made it possible to record interaction networks. Installed probes record events relating to user input, process management, communication, and use of important system objects, in particular the processor and the discs. Because INMON is a prototype, we have chosen not to instrument some kernel functions, with streams and TCP being the most significant omissions. We have, however, instrumented pipes, a stream style sockets in the Unix domain, and a network protocol (UDP/IP). Probes are installed to record events relating to NFS, the most important distributed service in SunOS.

Event recording is based on the approach used in the SunOS audit function. Event records are placed in buffers within the event recorder. A background process uses a system call to transfer event records from the buffers to a log file. Each node has an independent event recorder, with one log file produced for each node. This is the form in which the analysis programs discussed in the next chapter receive event records. The event recording mechanism is simple, and has worked well. Its overhead may be reasonably high, but INMON is a prototype designed to show the feasibility of identifying tasks and constructing interaction networks.

Two filtering mechanisms are provided. First, the only events recorded are those that occur in sub-tasks which have valid sub-task numbers, that is sub-tasks that are part of a task that resulted from a user input. Second, when the event recorder is started, a list of event types that are to be ignored can be specified. Note that events that are related to the event recorder are always ignored.

Some improvements could be made. For example, the sub-task number of message-related events could be stored in the sub-task number event-independent parameter. Event parameter choices could be evaluated, with unused parameters removed from the prototype. Also, other kernel functions could be instrumented. Experience indicates that the TCP/IP protocol and streams could be instrumented, based on techniques used for Unix domain stream sockets and the UDP protocol. System V semaphores and messages should also be straightforward to instrument, but shared memory instrumentation would be more difficult, probably requiring instrumentation to be added to application programs.

The INMON probes and event recorder, as they currently stand, are capable of recording a large range of tasks. As we will show in the discussion of analysis tools in the next chapter, the log files produced by the probes and the event recorder provide information from which interaction networks can be constructed and analysed.

Chapter 10

INMON: Analysis Tools

The analysis functions of INMON are performed by a number of separate programs, which operate on log files of two types: *node log files*, and *task log files*. A node log file is produced by the event recording program, `perfd`, as described in Subsection 9.5.2. A task log file contains all of the event records for a single task. Since a performance database has not been developed for INMON, a performance analyst must place log files into some form of directory hierarchy, and supply names of log files to analysis programs. This unsophisticated approach has been found adequate for the prototype implementation.

Two programs, to be described in Section 10.1, accept as input log files of both types. `perfdump` displays details of all event records in a log file using a textual "dump" format. `pfilter` produces an output log file by removing records of one or more event types from an input log file.

`insplit` (Section 10.2) extracts task log files from a set of node log files that were recorded at the same time, and all other analysis programs take input from task log files produced by `insplit`. `stripserver` (Section 10.3) removes from a task log file, some events that are not really part of a task. `analyse` (Section 10.4) calculates many different statistics based on information from one or more tasks, and in particular can give response time decompositions of time spent on the critical path of an interaction network. `toxgrab` (Section 10.5) provides a mechanism by which an interaction network can be displayed. From a task log file `toxgrab` produces an input file for `Xgrab`, a program that displays directed graphs. `inbrowser` (Section 10.6) also displays interaction networks, but in addition provides many analysis and filtering functions. The main difference between these two methods of display is that `Xgrab` "knows" only about directed graphs, so can only display an interaction network, whereas `inbrowser` "knows" about interaction networks, so can provide many features in addition to interaction network display.

Note that each task log file is stored as a pair of SunOS files. The first contains the event records, in the same format as that used for node log files. The second contains, for each event record in the first file, the number of the node on which the event record was recorded. The second file is necessary, as events within a single task are often recorded on more than one node. Node log files are stored as a single SunOS file, with the number of the node on which the event records were recorded stored in the log file header.

Unix manual pages for all programs described in this chapter are given in Appendix B.

10.1 Utility Programs

`perfdump` (Subsection 10.1.1) and `pfilter` (Subsection 10.1.2) can take as input both node and task log files. The former prints a textual description of each event record in a log file, and the latter removes event records of specified types from a log file.

10.1.1 perfdump

`perfdump` produces from a specified log file, a textual display of:

(1) the data in the header of the log file, and

(2) the data in each event record.

An example of the display for an event record is shown in Figure 10-1, with the event being the source event of a task, a terminal input received event.

```
PERF_TERM_INPUT_END:     size = 36        pid = 151        stn = 400130d
        seq_no = 0       pcnt = 3          time = 09:19:59.238593
        field old_stn:
                100bf03
        field dev:
                Major = 12, minor = 0
        field count:
                1
.inetnum:
        84 b5 0a 04
```

Figure 10-1: A textual description of an event record

The event name, PERF_TERM_INPUT_END in the example, is the first item shown. (As all INMON event names being with "PERF_", this prefix has been omitted from event names in this thesis). Then the values of the other event-independent parameters are printed: event record size, process pid, sub-task number, parameter count, and event timestamp. Also the position, numbered from 0, of the event record within the log file is printed.

Then the event-dependent parameters are listed. A TERM_INPUT_END event has three event-dependent parameters:

(1) the sub-task number that the process had before this event,

(2) the device that input was received from (in this case /dev/ttya, one of the serial ports), and

(3) the number of bytes received.

Finally, for task log files, the node on which the event was recorded is given, 132.181.10.4 (tui.cosc.canterbury.ac.nz) in this case.

10.1.2 `pfilter`

`pfilter` is a simple event filter, which takes as command line arguments the name of a log file and a list of event types. It produces a new log file, which is an exact copy of the input log file except that event records of the types specified on the command line have been removed. `pfilter` should be used only for removing simple event types as, if events of any other event types are removed, it will be impossible to construct interaction networks from the log file produced.

`pfilter` was originally developed to filter out large numbers of events relating to processor use that were recorded because of an initial problem with `perfd`. The problem resulted from the choice of conditions under which `perfd` was scheduled to write event records to a node log file (see part (2) of Subsection 9.5.1). Although filtering out these processor use events reduced the amount of information in a log file, it was more practical to display these interaction networks with the processor use events removed.

10.2 Producing Task Log Files

`insplit` takes as input a group of node log files that contain event records recorded on different nodes over the same period, and produces as output one task log file for each task found in the input log files. The node log files from which input is taken collectively contain event records for all events that were recorded on the nodes of a distributed system during a particular period. Performance information can then be extracted from the task log files by the programs to be described in Sections 10.4 to 10.6.

`insplit` proceeds through three major phases. First, a linked list of event records is created for each of the node log files. During this procedure some event records, in particular event records that contain a queue of sub-task numbers, are replaced by one or more event records. Second, the task source events are identified, and each event record is linked to its task source event. Finally, a task log file is created for each task identified. These phases are described in Subsections 10.2.1, 10.2.2, and 10.2.3.

10.2.1 Event Record Replacements

On input, most event records are simply appended to the list of event records being built-up for a node log file. Events records that contain a queue of sub-task numbers are, however, replaced by one or more event records, as described below.

(1) SOCK_SBREL

The SOCK_SBREL probe is executed whenever a socket buffer with a queue of sub-task numbers associated is released. The single event-dependent parameter of a SOCK_SBREL event record is the queue of sub-task numbers associated with the released socket buffer. When the socket buffer is released, the data is lost, so the sub-tasks associated with the data are terminated.

A SOCK_SBREL event record is replaced by a set of event records, one for each of the sub-task numbers stored in the queue. These event records are of a new type, namely SINK. Each SINK event record is constructed in the following way. Values of the event-independent parameters of process id and timestamp are copied from the SOCK_SBREL event record. The sub-task number event-independent parameter is set to a sub-task number from the queue, and the event number is set to SINK. A SINK event has as its only event-dependent parameter a `sink_info` structure. This structure has three fields, which record the sub-task number and the event number of the event record from which the SINK event record is created (a SOCK_SBREL event record in this case), and `id`, a number that can be set to a value useful in later analysis. For SINK events that result from a SOCK_SBREL event, `id` is 0.

(2) SOCK_STREAM_RCV

The SOCK_STREAM_RCV probe is executed whenever data is read from a receive buffer associated with a stream socket. The three event-dependent parameters of this event are: the number of bytes read, the sub-task number generated to accompany the data read, and a queue of N <sub-task number, length> pairs which show the origins of the data read. A SOCK_STREAM_RCV event is a message join/split event.

Each SOCK_STREAM_RCV event is replaced by a PE_DEQUEUE event and N-1 SINK events. A SINK event is generated for each of the first N-1 sub-task numbers in the queue. SINK events are generated as a precaution, because some of the sub-task numbers in the queue may be part of a task different from that of the Nth sub-task. Sub-tasks from these other tasks terminate at the SOCK_STREAM_RCV event, so generating SINK events for them is appropriate. The SINK events generated for the sub-tasks in the same task as the Nth sub-task can be handled appropriately by the other analysis tools, which must be able to associate the PE_DEQUEUE and the SINK events that result from a single SOCK_STREAM_RCV. Therefore, a sequence number is maintained, that is incremented for every SOCK_STREAM_RCV. SINK events are generated in the same way as in (1), and the current value of the sequence number is assigned to the `id` field of the `sink_info` structure.

A PE_DEQUEUE event record is constructed in the following way. Values of the event-independent parameters of process id, sub-task number, and timestamp are copied from the SOCK_STREAM_RCV event record. Note that SOCK_STREAM_RCV is a message-related event, so the value of the sub-task number event-independent parameter in a SOCK_STREAM_RCV event record will always be 0. The event number is set to PE_DEQUEUE. A PE_DEQUEUE event has three event-independent parameters. The first is set to the Nth sub-task number in the queue, and the second to the sub-task number of the message read (these two parameters contain the two sub-task numbers that each fork event must contain). The third parameter is a `sink_info` structure. This structure has three fields, which record the event number of the event record from which the PE_DEQUEUE event record was created (a SOCK_STREAM_RCV event record in this case), the length of the data read, and `id`, a number that can be set to a value useful in later analysis. The `id` field is set to the current value of the sequence number described above, thus allowing related PE_DEQUEUE and SINK events to be identified.

To summarise, each SOCK_STREAM_RCV event record is replaced by a PE_DEQUEUE event in the Nth sub-task, and SINK events in the N-1 other sub-tasks. This is done as a precaution in case the sub-tasks that join at a SOCK_STREAM_RCV are part of two or more different tasks. See part (1) of Subsection 10.4.1 for further discussion of PE_DEQUEUE and SINK events.

10.2.2 Assigning Event Records to Tasks

After the event records from each node log file have been read into a linked list, and the manipulations described in Subsection 10.2.1 performed, an attempt is made to associate each event record with a task. A tree of event records (as described in part (1) of Subsection 7.3.3) is created for each task found.

10.2.3 Producing Task Log Files

Finally, a task log file is created for each task. To create a task log file, the tree of event records of a task is traversed, and the event records are written, in such a way that:

(1) The event records for each sub-task are stored contiguously, and are in increasing order of time of occurrence.

(2) Event records for each sub-task, other than the first sub-task, appear in the task log file after the fork event record that marks the creation of the sub-task.

The only useful piece of information stored in the header of a task log file is the time at which the log file was created by `insplit`. The TRAILER record of a task log file contains the sum over all input node log files of the number of dropped event records.

All of the other analysis tools, described below in Sections 10.3 to 10.6, operate only on task log files.

10.3 Removing Extraneous Event Records

In Subsection 7.3.2, it was observed that it may be reasonable to assume that a thread finishes working for a sub-task, when a thread performs an operation that results in it becoming associated with another sub-task. This assumption, with one exception, has proved to be valid for INMON.

The exception is the result of the way in which `nfsd` processes operate. Every node that operates as an NFS server executes a number of `nfsd` processes, usually eight, that are responsible for performing remote procedure call requests made from NFS client nodes. When an NFS request arrives, all `nfsd` process currently waiting for a request are put onto the processor queue. The first `nfsd` process to run handles the request, and the remainder resume waiting.

Let us now consider the sub-task numbers assigned to `nfsd` processes. When an `nfsd` process receives a request message, the sub-task number associated with the incoming message is assigned to the `nfsd` process. The process then performs the request, sends the reply, then waits for the next request to arrive. The sub-task number of the process is still the one received with the request message. A problem arises if the process is then awoken several times only to find each time that no request message is available. Because no request message is available, events relating to processor use are recorded that have the sub-task number of the last request, when in fact the `nfsd` process finished work for that sub-task at the time that it waited for another request, immediately after sending the reply.

The solution adopted was to write a program, `stripserver`, that removes these extraneous event records from a task log file. `stripserver` produces a version of a specified task log file, from which all records in a sub-task that appear after a RPCSVC_WAIT record have been removed. An alternative would be to change the RPCSVC_WAIT probe, so that after the event is recorded the sub-task number of the process is set to PE_INVALIDIPID.

Usually, the analysis programs described in Sections 10.4 to 10.6 take as input task log files that have been produced by `stripserver`.

10.4 Calculating Statistics

`analyse` has been developed to calculate statistics from a group of one or more task log files. Its major outputs are a decomposition of response time with respect to sub-task states

(as defined in Section 6.1 and Subsection 6.2.4), some basic statistics (see Subsection 6.2.3), and information that allows checks to confirm the accuracy of INMON. A very important output is a decomposition of time spent on the critical path through an interaction network.

`analyse`, and `toxgrab` and `inbrowser`, which will be described in Sections 10.5 and 10.6, operate on one or more interaction network data structures, each of which is created from the event records in a task log file. These data structures, and way in which they are constructed, are discussed in Subsection 10.4.1. The different levels of aggregation at which results can be reported (see Subsection 6.2.1) are described in Subsection 10.4.2. The actual output reports produced are described in Subsection 10.4.3.

10.4.1 Creating an Interaction Network Data Structure

The interaction network data structure is established in two steps. (For the remainder of this chapter, interaction network data structures will be referred to as interaction networks.) In the first step, the event records of a task log file are read, and an interaction network created from them. In the second step, the critical path through the interaction network is marked.

(1) Loading the interaction network

Loading an interaction network from a task log file is done in three phases.

(1) a tree of event records is created.

(2) additional links are added.

(3) some event records are removed.

In the first phase, a list of event records is created for each sub-task. The first event of the first sub-task is the source event. For the first event of every other sub-task, a link is created from the fork event that created the sub-task, to the first event in the sub-task.

In the second phase, *join* links are added to the tree constructed in the first phase, thereby turning it into an interaction network. The following types of join links are added:

(i) For ordinary join events, a link is added from the last event in the terminating sub-task to the join event.

(ii) For SINK events which have matching PE_DEQUEUE events, the SINK event is removed, and the link to the SINK event is changed to point to the PE_DEQUEUE event (see part (2) of Subsection 10.2.1).

In the third phase, some event records that appear at the end of some sub-tasks, but which really belong to other tasks, are removed. An example of a situation that can give

rise to these extraneous events is shown in Figure 10-2. In the figure, a process waits for and then receives a message. If the sub-task of the process before receiving the message is st_1, and the sub-task of the message is st_2, then events 1, 2, 4, and 5 are recorded as being part of st_1, and events 3 and 6 are recorded as being part of st_2. Events 4 and 5 are in reality associated with st_2, as they represent the process being rescheduled in order to receive the message.

The event records removed in the third phase of loading an interaction network, occur where st_1 and st_2 are associated with different tasks, say t_1 and t_2. In these cases the final events in st_1 are actually associated with another task, with a potentially large gap between the last event in st_1 related to t_1, and the events in st_1 related to t_2. A problem that can occur is that the last event in st_1 (an event that actually occurred in t_2) might be mistakenly chosen as v_f. This is quite likely where user input occurs. Here, there is no joining sub-task; event 6 is associated with a task newly created to process the user input. The scheduling events, events 4 and 5, are recorded for the sub-task previously associated with the reading process, potentially quite some time after the process had finished work for the sub-task.

The best solution would be to associate these scheduling events with the correct sub-task, by making the SETRQ a join vertex in these situations. Where the critical path was associated with the message, this would allow the critical path to include the process

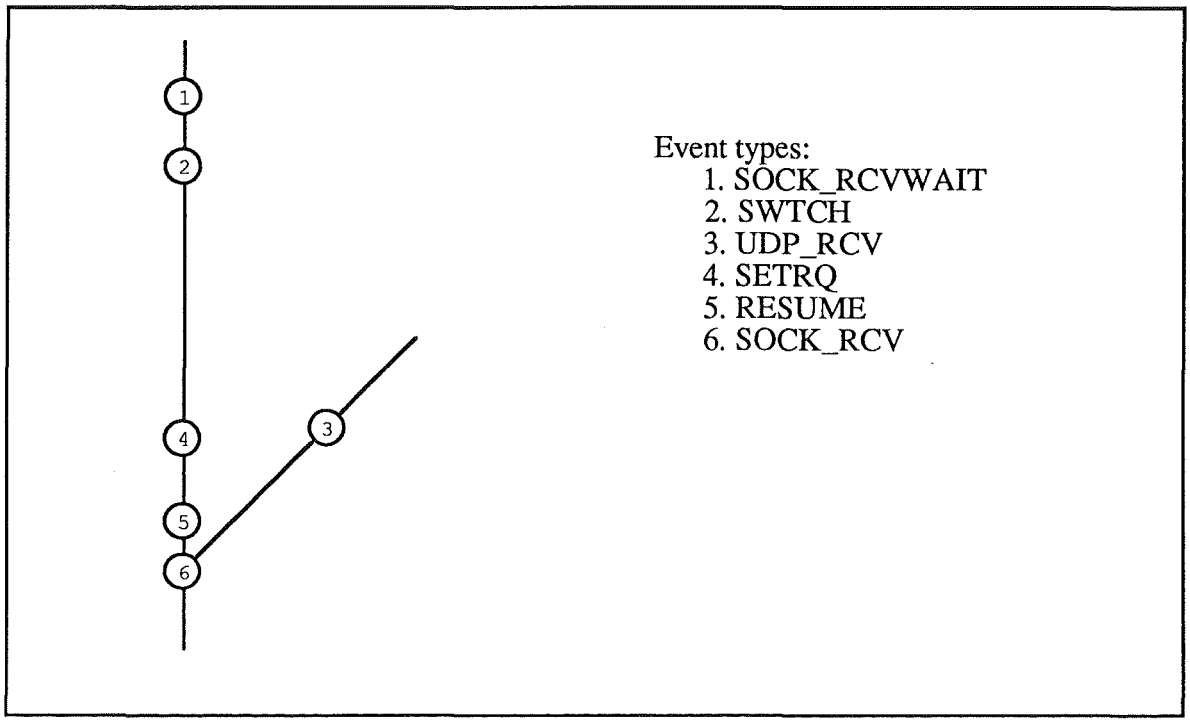


Figure 10-2: Illustration of how extraneous events can arise

scheduling events. In the meantime, SETRQ and RESUME event records that occur immediately prior to a sink vertex are removed from every sub-task.

(2) Determining the critical path

The critical path is determined using the algorithm given in Figure 5-2. First, the event record with the greatest timestamp is selected as v_f. Then the critical path is traversed in reverse, from v_f to v_0, with each event record on the path marked as being on the critical path. If a join vertex is encountered with two non B-edges incident on it, then the edge followed in reverse is the one that represents an activity in the process in which the join event was recorded. Also, a warning message is printed informing the user that the interaction network contains multiple critical paths.

During the experiment described in Section 11.3, it was found that because process dependencies resulting from use of some shared resources are not recorded by INMON, the critical path determined by INMON is occasionally inaccurate. The inaccuracy in the experiment described in Section 11.3 resulted from a period of intense context switching between two processes in the same task. This problem can be solved by having INMON recognise process dependencies that result from use of shared resources.

10.4.2 Levels of Aggregation

`analyse` provides, at the three levels of aggregation described in Subsection 6.2.1, a number of reports on process-related activities and on message-related activities. At the lowest level, the *process* level, summaries are provided on the activities of each process, and on message passing between each pair of processes. At the middle level, the *node* level, summaries are provided on all process-related activities on each node, and on message passing for each pair of nodes. At the highest level, the *task* level, one summary is provided of all process-related activity in the task, and one summary is provided of all message-related activity in the task.

Note that where communication between two processes or between two machines proceeds in both directions, performance information is presented separately for the messages sent in each direction.

10.4.3 Reports Produced

`analyse` can produce twelve reports from its input set of task log files. Each report can be specified by a unique combination of the values of the variables A, P, and CP, described below.

(i) A represents the level of aggregation: process, node, or task.

(ii) P is true if the report is for process-related activities, false if the report is for message-related activities.

(iii) CP is true if the only activities included in the report are those that are on the critical path, and false if all activities are included.

For example, the report <A = node, P = false, CP = true> summarises for each pair of nodes <N_A, N_B>, the message passing activities that are on a critical path (of an interaction network being analysed) and that were involved in sending messages from Node N_A to Node N_B. The user can, each time that `analyse` is run, select which of these twelve reports is to be produced.

Each report contains some number of sections. Task level reports contain one section. A node level report on process-related activities contains one section for every node that performed part of the task. A node level report on message-related activities contains one section for each pair of nodes <Node N_A,Node N_B>, N_A and N_B not necessarily distinct, such that during the task a process on Node N_A sent messages to a process on Node N_B. A process level report on process-related activities contains one section for each process that performed part of the task. A process level report on message-related activities contains one section for each pair of processes <P_A, P_B>, such that during the task Process P_A sent messages to Process P_B.

Each report section contains some number of subsections. Reports on process-related activities contain five subsections: event counts, sub-task state counts and times, disc usage, RPC statistics, and validation. Reports on message-related activities contain only the event counts, and sub-task state counts and times subsections. The five subsection are described below in (1) to (5).

The examples to be given in (1) to (5) all come from analysis of the same task log file. The task was recorded in the environment described in Subsection 7.5.1 and Section A.1. A user is typing on a terminal connected to tui. Many systems programs used by tui are stored on weka, including the program for the `ls` command. The user input for the example task is the typing of the newline character, which has been preceded by the 'l' and the 's' characters. The input is received by a shell process, which creates a new process to run the `ls` (list directory files) command. The shell process waits for the `ls` process to finish, and when it does finish the shell prints its prompt and waits for the next input. Both the shell process and the `ls` process make NFS remote procedure calls to weka, and, in this example, all of these requests are performed by the same `nfsd` process on weka. Figure 10-10 shows the interaction network for the task, and Figure 10-1 gives a description of the first event recorded for this task.

The examples used in (1) to (4) are all from reports at the task level of aggregation. The full set of twelve reports generated from the task log file used for the examples in this chapter, is included as Appendix C.

(1) Event counts

The event counts are simply counts of the number of events of different types that occurred in the set of activities being aggregated. At the lowest level, each event that occurred in a process is counted for that process, and the message-related events are counted for the pair of processes <P_A, P_B> that the message is passed between. At the higher levels, the event counts are simply the totals of event counts over some number of lower level components.

In Figure 10-3, the process event counts are shown for all events (the left hand column), and for all events on the CP (the right hand column). Notice that most events are on the critical path, indicating that there was little concurrency in the performance of the task. This is to be expected for the SunOS environment. All of the RPC call, send, and receive events are on the critical path, but none of the RPC wait events are. Also, the event of the shell process starting to wait for the `ls` process to finish, is not on the critical path, as the critical path branched off into the `ls` process before the PERF_SIGPAUSE_START event occurred in the shell process.

```
Event counts:                          Event counts:
       1  PERF_FORK                           1  PERF_FORK
       1  PERF_EXIT                           1  PERF_EXIT
       1  PERF_ZOMB_EXIT                      1  PERF_ZOMB_EXIT
       1  PERF_TERM_INPUT_BEGIN               1  PERF_TERM_INPUT_BEGIN
      14  PERF_RPCCLNT_SEND                  14  PERF_RPCCLNT_SEND
      14  PERF_RPCCLNT_RECV                  14  PERF_RPCCLNT_RECV
      14  PERF_RPC_CALL                      14  PERF_RPC_CALL
      69  PERF_SWTCH                         52  PERF_SWTCH
      69  PERF_RESUME                        52  PERF_RESUME
      69  PERF_SETRQ                         52  PERF_SETRQ
      14  PERF_SD_STRATEGY                   14  PERF_SD_STRATEGY
      14  PERF_BIOWAIT_END                   14  PERF_BIOWAIT_END
      14  PERF_BIOWAIT_START                 14  PERF_BIOWAIT_START
       1  PERF_EXEC                           1  PERF_EXEC
       1  PERF_SIGPAUSE_START
       1  PERF_SIGPAUSE_END                   1  PERF_SIGPAUSE_END
      14  PERF_RPCSVC_WAIT
      14  PERF_RPCCLNT_WAIT
       1  PERF_TERM_INPUT_END                 1  PERF_TERM_INPUT_END
      14  PERF_RPCSVC_SEND                   14  PERF_RPCSVC_SEND
      14  PERF_RPCSVC_RECV                   14  PERF_RPCSVC_RECV
```

Figure 10-3: Event count subsections for process events

In Figure 10-4, the message event counts are shown. This subsection is the same for the report that counts all events, and for the report that counts only events on the critical path. Twenty-eight UDP datagrams were sent, two for each RPC. Note that the message send and message receive events are process-related, so are counted only in process reports.

```
Event counts:
      28   PERF_UDP_RCV
      28   PERF_UDP_SEND
```

Figure 10-4: Event count subsections for message events

(2) Sub-task state counts and times

The sub-task state counts simply report the number of activities recorded for each of the states described in Subsection 9.2.5. The sub-task state times provide details of time spent in each state. The state times are one of the most important of the outputs of `analyse`, with state time breakdowns for activities on a critical path of particular importance, because they give the components of response time. In INMON, there are only sixteen sub-task states, so it is reasonable to simply give the time spent in each of the states, and the more sophisticated analysis techniques described in part (1) of Subsection 6.2.4 are not needed.

In Figure 10-5, the process sub-task state counts and times are shown for all events (the left hand column), and for all events on the critical path (the right hand column). Most of the `run`, `ready`, and `sleep_disc` times are on the critical path, and most of the other times are not. In Figure 10-6, the message sub-task state counts and times are shown, and again the subsection is the same for both the all events, and the critical path events reports. All of the message-related activities happen to be on the critical path of this task.

The critical path length for a task, and therefore its response time, is the sum of the process-related and message-related components of the critical path, which in the example is 1.93s. The major components of the critical path are: time spent running on or waiting for the processor (0.89s); time spent waiting for disc transfers to complete (0.64s); and time spent sending RPC messages (0.39s).

By using reports at lower levels of aggregation, it is possible to perform more detailed decompositions. The critical path consists of 1.49s on tui, 0.05s on weka, 0.11s for messages travelling from tui to weka, and 0.27s for messages travelling from weka to tui. The message time from weka to tui is more than twice that of the message time for the other direction because, as we will show in (4), most messages from tui to weka are read request

messages (small), whereas most messages from weka to tui are read replies (large, as they contain the data read). The total time spent on each node, and the total message time between two nodes, can in turn be decomposed. The time spent on tui, for example, consists largely of 0.83s spent in the `run` and `ready` states, and 0.64s spent in the `sleep_disc` state.

Further decomposition can be achieved by using process level reports. For example, the shell process occupies 0.92s of the critical path on tui (0.45s processor, 0.47s disc), and the `ls` process 0.56s (0.39s processor, 0.17s disc).

```
State counts:                          State counts:
      239  run                              177  run
       69  ready                             52  ready
        1  created                            1  created
        1  zombie
        1  sleep_child_wait
       14  sleep_RPC_clnt
       14  sleep_disc                        14  sleep_disc
        2  sleep_other                        1  sleep_other

State Times:                           State Times:
      0.88294  run                           0.62285  run
      0.33516  ready                         0.26833  ready
      0.00379  created                       0.00379  created
      0.07998  zombie
      0.48761  sleep_child_wait
      0.35336  sleep_RPC_clnt
      0.64295  sleep_disc                    0.64295  sleep_disc
      0.20404  sleep_other                   0.00991  sleep_other
```

Figure 10-5: Sub-task state counts and times subsections for process events

```
State counts:
       57  queued
       28  network

State Times:
      0.13162  queued
      0.25741  network
```

Figure 10-6: Sub-task state counts and times subsections for message events

(3) Disc usage

The subsection on disc usage provides, for each disc partition, a detailed breakdown of read and write access counts and times. Each SCSI disc attached to a node is identified by

a small integer, unique within that node. Each disc can be divided into up to eight partitions, with each partition identified by an integer in the range 0 to 7.

In the reports aggregated to the task level, access counts and times reported for, say, drive 0, are the totals of access counts and times over all nodes on which accesses to a local drive 0 were recorded. Disc access summaries for each node are available in node level reports, and for each process in process level reports.

The subsection on disc usage at the task level is shown in Figure 10-7. All disc accesses were on the critical path, so the disc usage subsection is the same for both the report on all events and activities, and the critical path report. Using the node level reports, the disc accesses performed on each node can be determined. All of the disc accesses reported in Figure 10-7 occurred on tui. This means that none of the fourteen NFS RPCs made to weka required any disc accesses, showing that the information needed was already in weka's main memory. The fact that there were fourteen RPCs made, and fourteen disc accesses, is a coincidence.

Using the process level reports, we can identify the number of disc accesses made by each process. In the example, the shell process performed the twelve disc reads on partition 1, drive 0 on tui, and the `ls` process performed the other two disc reads, one from partition 0, and one from partition 6. Note that partition 0 is the root filesystem on tui, that is the root of tui's directory hierarchy, partition 1 is the swap partition, that is the backing store for virtual memory, and partition 6 contains user files.

```
Disc usage:
      Drive 0
      Access counts:      Part.   Reads  Writes   Total
                             0       1       0       1
                             1      12       0      12
                             6       1       0       1
                         Total      14       0      14
      Access times:       Part.      Reads     Writes      Total
                             0    0.03435    0.00000    0.03435
                             1    0.46873    0.00000    0.46873
                             6    0.13987    0.00000    0.13987
                         Total    0.64295    0.00000    0.64295
```

Figure 10-7: Disc usage subsection

(4) RPC statistics

The subsection on RPC statistics provides a detailed breakdown of the remote procedure calls made by the kernel. Each program to which remote procedure calls are made is identified by a program number and a version number. Each procedure within the

program is identified by a small integer. Currently, `analyse` reports these numbers rather than program and procedure names, although it would not be difficult to produce names rather than numbers.

The subsection consists of two parts, one on calls made by clients, and the other on calls performed by servers. Client statistics include, for each remote procedure called: the number of calls made to that procedure; and the total amount of time a client was delayed waiting for calls to that procedure to finish. These statistics on call times do not appear in critical path reports, as there is no time spent on a critical path waiting for a remote procedure call to finish. The server statistics include, for each remote procedure call called, the number of calls received for that procedure. Counts and times for remote procedures are grouped by program.

The subsection on RPC statistics from the task level report on all events is shown in Figure 10-8. The equivalent subsection from the critical path report is the same, except that it lacks the call times. At the task level, the counts and times for all procedures called by clients, and the counts for all procedures performed by servers, are shown. In the example, all calls were to program 100003, version 2, which is the NFS program. Two calls were made to procedure 1, (get file attributes), one call to procedure 5 (read symbolic link), and eleven calls to procedure 6 (read data).

Again, more detailed information is available from reports at lower levels of aggregation. From the node level reports, it is apparent that all calls were made from processes on tui to processes on weka. From the process level reports, it is apparent that eight of the read data calls were made by the shell process on tui, and the other six calls

```
RPC Client statistics
      Program 100003, Version 2
            Call counts
                  2  Proc. 1
                  1  Proc. 5
                 11  Proc. 6
            Call times
             0.02566  Proc. 1
             0.00998  Proc. 5
             0.31772  Proc. 6

RPC Server statistics
      Program 100003, Version 2
            Call counts
                  2  Proc. 1
                  1  Proc. 5
                 11  Proc. 6
```

Figure 10-8: RPC subsection of the all events report

were made by the `ls` process on tui. Also, all calls were received and performed by a single `nfsd` process on weka.

(5) Validation

Subsections containing validation information appear only in the reports where <A = process, P = true, CP = false>, that is reports on all process-related events, at the process level of abstraction. In these reports, the validation subsection is produced only for processes where all the events of the process are recorded in the interaction network, from the FORK that created the process, to the ZOMB_EXIT that finally terminated it. A validation subsection provides comparisons between information on the resource usage of a process as computed by the kernel, and as computed by `analyse` from information on the process in the interaction network. Any disparities between the two sets of figures would indicate that either the kernel or INMON was not operating correctly.

In Figure 10-9, the validation subsection is shown of the `more` process of the task whose interaction network is shown in Figure 11-5. The left hand column contains information computed by `analyse`, and the right hand column contains the same information as computed by the kernel. In Figure 10-9, all values computed by INMON are consistent with those computed by the kernel.

```
Process validation
      Quantity          Monitor          should be        Rusage
      CPU run time          3.32013      = (approx)           3.46000
      vol cswtch                 53      = (exact)                 53
      invol cswtch              110      = (exact)                110
      Msg recvs                   0      = (exact)                  0
      Msg sends                  59      <=                        59
      Swaps                       0      = (exact)                  0
```

Figure 10-9: An example of a validation subsection

The validation subsection consists of six comparisons:

(i) CPU time. The kernel keeps a record of the amount of processor time that a process uses, divided into time in user mode, and time in kernel mode. `analyse` calculates the total amount of processor time used by a process as the time it spends in `run` state. The kernel uses a sampling technique to record processor time use, with a time between samples of 20ms. The event records in an interaction network record the times at which processor use begins and ends, with microsecond resolution. The processor times recorded by each method should be comparable, although the time calculated by `analyse` should be the more accurate.

(ii) voluntary context switches, and (iii) involuntary context switches. When a process gives up the processor, it may do so voluntarily because it has reached a point in its execution where it must wait for some event before it can continue, or involuntarily because the processor is to be allocated to a higher priority process. The kernel keeps counts for each of these types of context switch for each process. In an interaction network, there is a SWTCH event record for every context switch that occurred. Also, `analyse` can distinguish between the two types of context switches. When an involuntary context switch occurs, the process is still ready to run, so it is put back on the processor list (recorded by a SETRQ event record) before the context switch occurs. With voluntary context switches, the SETRQ event occurs some time after the SWTCH. The numbers calculated by both methods should match.

(iv) Message receives. The kernel maintains for each process a count of the number of successful socket reads. A SOCK_RCV event should be recorded whenever the counter is incremented, so the number of SOCK_RCV events recorded for a process should match the number of message receives recorded by the kernel. Note that in Figure 10-9 the number of message receives recorded for the `ls` process is 0, despite the fact the the `more` process exchanged messages with an `nfsd` process on weka. Kernel remote procedure call messages do not, however, pass through the functions that maintain the message send and receive counters.

(v) Message sends. The kernel maintains for each process a count of the number of attempts to write to a socket. The SOCK_SEND probe is executed whenever data is successfully sent. Usually, one SOCK_SEND event is recorded every time the message send counter is incremented, but there are two situations in which this does not occur.

First, for stream sockets a message may be appended to a socket send buffer in several pieces. If this happens, the message send counter is increased by one, but more than one SOCK_SEND event is recorded. To detect this, a sequence number that is only incremented when the kernel send counter is incremented, is recorded as one of the event-dependent parameters of SOCK_SEND. The number of SOCK_SEND events with different values of the sequence number is computed for use in the validation subsection, rather than the number of SOCK_SEND events.

Second, the message send counter is incremented for all attempts to send a message, whereas SOCK_SEND events are only recorded on successful sends. The most common send error is where a connected stream socket has been closed, resulting in the send failing and a SOCK_EPIPE event being recorded. To produce a count comparable to the kernel message send count, `analyse` adds the SOCK_EPIPE event count to the number of SOCK_SEND events with different values of the sequence number. This number is nearly

always the same as the message send count of the kernel, but the kernel count may be greater if an unusual error occurs.

(vi) The number of times a process is swapped out. The kernel maintains this count for each process, and the count should match the number of SWAPOUTBEGIN events recorded for a process.

Not all items of resource use information recorded by the kernel are used in producing validation subsections. The kernel records some statistics relating to memory use, but INMON does not directly record memory use, so no comparison is possible. The kernel also maintains counts for input and output operations. In many cases a disc access or NFS RPC event is recorded by INMON when one of these counters is incremented, but in other cases this does not occur, so it is not possible to make a useful comparison. Finally, the kernel records the number of signals a process receives, but INMON does not (at present) record signal events.

10.5 Display of an Interaction Network

Although `analyse` provides a great deal of information, we felt that a graphical representation of an interaction network would provide additional information on a task's behaviour. Communication patterns can, for example, be readily observed from display of an interaction network. Rather than writing a display program from scratch, we decided to use a program designed to display directed graphs. In this approach, the only INMON program required is one that produces from a task log file an input file for the display program.

`Xgrab` [BARN89], to be described in Subsection 10.5.1, was selected as the display program most suitable for displaying interaction networks. `toxgrab`, detailed in Subsection 10.5.2, was written to produce Xgrab files from task log files. An example is given in Subsection 10.5.3, of an interaction network displayed by `Xgrab`, based on an Xgrab file produced by `toxgrab`.

This implementation provides the most basic of the browsing functions described in Subsection 6.2.8: we can display an interaction network, zoom and pan around the network, and highlight the critical path. `inbrowser`, to be described in Section 10.6, performs many of the more advanced functions described in Subsection 6.2.8.

10.5.1 `Xgrab`

`Xgrab` is an adaptation to the X Windows environment of `sungrab` [ROWE87], which runs in a Sunview environment. `Xgrab` was designed for display, edit, and layout of

arbitrary directed graphs. Only the display function of `Xgrab` is used for interaction networks, as editing is not required and layout is performed by `toxgrab`.

Features of `Xgrab` useful for displaying interaction networks are:

(i) There is no limit to the canvas size. This feature was the most significant for the selection of `Xgrab` ahead of other graph browsers, as most other graph browsers have a limited canvas size, often a single A4 page for instance. When `Xgrab` first displays a graph, all of the graph is displayed in a window, with the graph scaled to fit. The user can then use zooming and scrolling to look more closely at areas of interest in the graph.

(ii) `Xgrab` has a simple file format. An Xgrab file is a text file, with one line describing each vertex, and one line describing each edge. The major pieces of information required to describe each vertex are an identifier, a name with which the vertex can be labelled, and its X and Y coordinates. The major pieces of information required for each edge are the identifiers of its source and destination vertices, and the line type for the edge.

(iii) Several line types are available for edges, enabling edges on the critical path to be distinguished from edges not on the critical path.

10.5.2 `toxgrab`

`toxgrab` produces an Xgrab file from an interaction network. One vertex line is written to the Xgrab file for every vertex in the interaction network, and one edge line is written for every edge in the interaction network. `toxgrab` determines the coordinates and identifier assigned to each vertex, and the line type assigned to each edge. The identifier assigned to a vertex is the position in the task log file of the event record that the vertex represents. The line type for an edge can be easily determined from the critical path information present in an interaction network, with edges on the critical path drawn as dotted lines, and all other edges drawn as solid lines.

The major activity performed by `toxgrab` is the assignment of coordinates to each vertex. In the layout used by `toxgrab`, which is based on that described in Subsection 4.2.5, there is one column for every process in the task. X-coordinates of vertices are determined in the following way. Each vertex that represents a process-related event appears in the column of the process in which the event occurred. Time increases down the page, with events in each column appearing in order of occurrence. Vertices that represent message-related events appear somewhere between the vertices that represent the sending of the message and its reception.

We have made available two methods for determining vertex Y-coordinates. In the *time* layout, the Y-coordinate of each vertex is computed in proportion to the timestamp of the event record that the vertex represents. In the *fixed* layout, every vertex is some fixed

distance below the lowest of its preceding vertices. The time layout shows the relative times of events, but vertices representing events in a sub-task that occur very close (in time) to each other can be hard to distinguish. The fixed layout shows each vertex distinctly, but does not preserve timing information.

A heading vertex is produced for each process column. The vertex is positioned at the top of the process column, and is labelled with a number that contains part of the network number of the node on which the process executed, and the process pid.

10.5.3 An Example

The interaction network in Figure 10-10 is for the task described in Subsection 10.4.3. The time layout has been used. The figure is shown exactly as it appears on the screen, except that the process headings have been changed from numbers to names. Some points are:

(1) The time layout shows well the relative times at which events occurred. For example, the time at which each NFS request was made is clear. Also, it is obvious that for most NFS requests the reply took longer to send than the request. This is because, as noted above, most calls are to the NFS read procedure, which has a small request message and a large reply message .

(2) There is a considerable delay between when the `csh` process forks off the `ls` process, and when the `csh` process is subsequently placed on the processor queue. Shortly after the `csh` process does run again, it blocks waiting for the `ls` process to finish. From the definition of the `fork` system call in Subsection 9.1.1, the `csh` process should resume immediately after the `fork`. The reason that the `csh` process is delayed for so long, is that `csh` uses the `vfork` system call (described in part (4) of Subsection 6.1.3), rather than the `fork` system call. The considerable delay is the period during which the parent has loaned its address space to the child.

Note that an edge should be included in the interaction network, indicating that the child had informed the parent that it can continue. This enhancement would require no more than the addition of two probes to record the sending and the receiving of this "message".

This apparent anomaly is highlighted because the time layout was used. Also, the 0.194s spent off the critical path in the `sleep_other` state (see Figure 10-5) is the time during which the `csh` process is suspended in the `vfork` system call.

(3) Figure 10-10 shows the whole interaction network, and individual events are difficult to identify. Individual events are much more easily seen by zooming in to display only small areas of an interaction network. In the time layout, some events are tightly

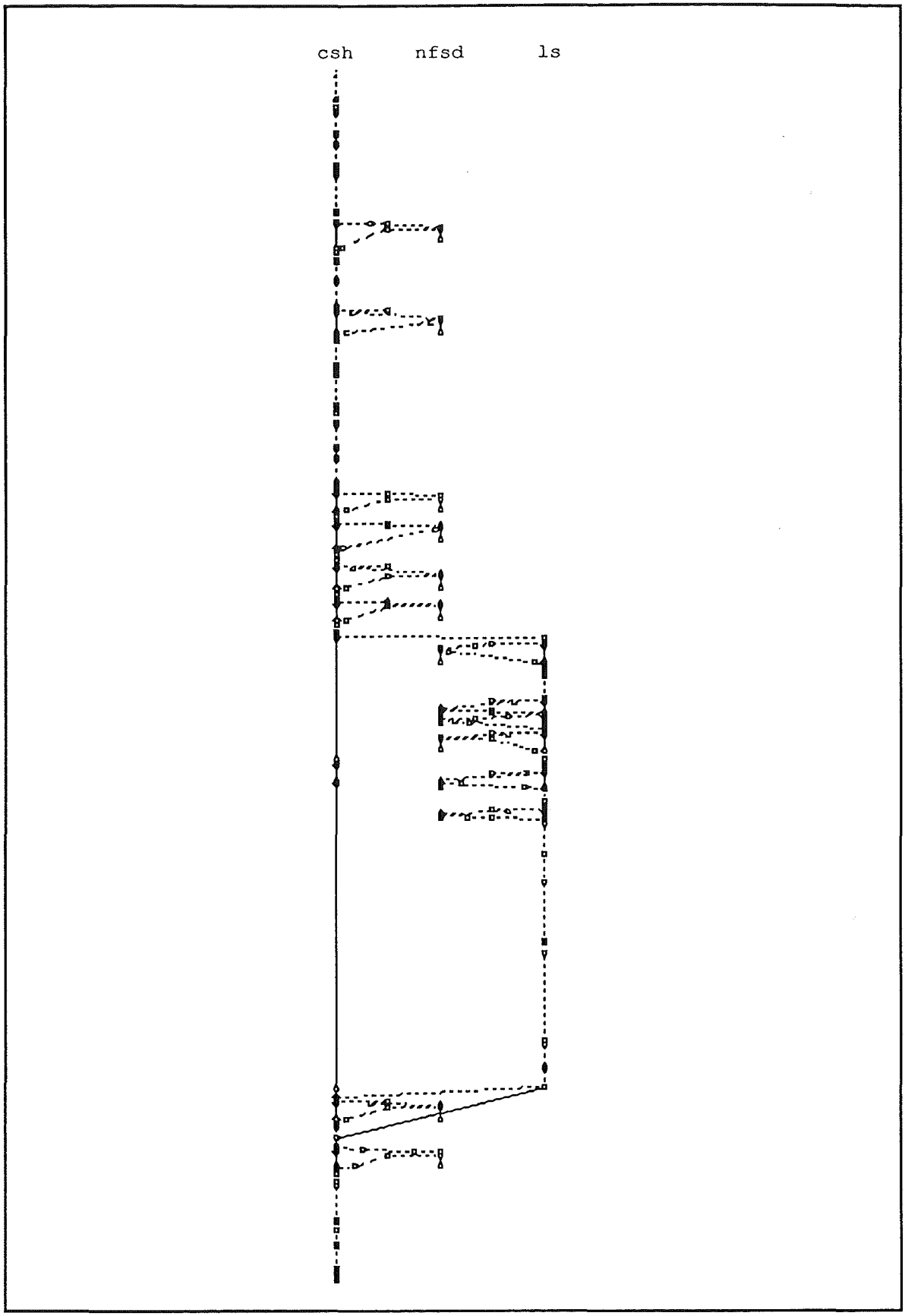


Figure 10-10: Interaction network for an execution of the `ls` command

grouped even at high levels of magnification, because the events occur very close together. In such situations the fixed layout can be used to allow identification of individual events.

(4) The critical path is clearly shown, with edges on the critical path represented by dotted lines. For instance, Figure 10-10 agrees with Figure 10-3 in showing all remote procedure call requests and replies as being on the critical path.

(5) Two edges from the last vertex in the `ls` process are incident on different vertices in the `csh` process. The first, the one on the critical path, represents the `csh` process receiving a "death of a child" signal notifying it that the `ls` process has finished. The second edge represents the time that the `ls` process spends in zombie state, with the join event in the `csh` process a ZOMB_EXIT event that finally lays the `ls` process to rest.

(6) Figure 10-10 helps to reinforce the fact that a task is very different from the execution of a program. The task shown in the interaction network contains a single period of the execution of the `csh` program, the entire execution of the `ls` program, and fourteen periods of execution of the `nfsd` program.

10.6 Browsing through an Interaction Network

An interaction network browser, `inbrowser`, has been implemented by Thoo Lip Chau under the author's supervision [THOO91]. `inbrowser` performs many of the functions discussed in Subsection 6.2.8. The basic aim behind `inbrowser` is the same as that of `Xgrab` and `toxgrab`, that is to display an interaction network. Since `inbrowser` "knows" about interaction networks, it can perform many functions that `Xgrab` / `toxgrab` cannot. Important features of `inbrowser` are:

(1) The Y-coordinate calculation method can be switched between time layout and fixed layout at any time. Also, the critical path edge type can be switched between dotted and solid at any time.

(2) Any of the reports produced by `analyse` can be produced and viewed.

(3) Event record descriptions produced by `perfdump` can be viewed. An individual event record can be selected for viewing using the mouse, or a dialogue can be used to select groups of event records based on criteria such as event type, the node an event was recorded on, and the process an event was recorded for.

(4) A filter can be created to exclude event records by event type, or by the node an event was recorded on, or by the process an event was recorded for. It is possible to have the display show only the records outside the current filter, and to specify that the analyse and record dump utilities deal only with records outside the current filter.

10.7 Summary

In this chapter, the analysis tools developed as part of INMON have been discussed. The most important of these tools are `insplit` and `analyse`, and the programs for displaying interaction networks, `Xgrab` and `toxgrab`, and `inbrowser`. `insplit` produces task log files from a set of node log files. The task log files can then be analysed using the other programs.

`analyse` can give summary reports at three levels of aggregation, namely the task level, the node level, and the process level. Separate reports are produced for process-related events and activities, and for message-related events and activities. Also, reports can summarise information for all events and activities, or for the events and activities on the critical path only. One of the most important pieces of performance information available from reports produced by `analyse`, is a detailed breakdown into state times of the time spent on the critical path. An overall decomposition at the task level can be examined first, followed by more detailed decompositions at the node and process levels. Detailed summaries of the disc and RPC activities are also available. The validation summaries show the accuracy of the INMON information, by providing comparisons with information collected by the kernel.

The display of interaction networks available from `Xgrab` and `inbrowser` can provide a performance analyst with valuable information. In particular the overall structure of a task is very easily seen. `inbrowser` combines the ability to display an interaction network with: a filtering function, and the ability to produce the reports of `analyse`.

Although possible improvements to these analysis tools can be seen, the INMON tools do show that interaction networks can be identified, and that very useful performance information can be extracted from them, in the form of reports and displays. The value of the tools is demonstrated further by the experiments described in the next chapter.

Chapter 11

Experiments

Four experiments performed using INMON are described in this chapter. The experiments have been chosen to show a wide range of the capabilities of INMON. In the first three, the execution of individual interactions are examined, and in the fourth a set of interactions is analysed. A brief summary of the experiments follows.

Experiment 1: Compilation and linking of a simple C program

Two compilations are analysed, one where the `-pipe` option is specified, and one where it is not.

Experiment 2: Two processes communicating using a pipe

Two interactions are analysed, one with file readahead enabled and one with it disabled.

Experiment 3: A window resize interaction

For this interaction, the user input is a mouse button up event, rather than a keystroke.

Experiment 4: A set of interactions

The set of interactions analysed is that used in the multi-user benchmark of the MUSBUS Unix benchmarking package.

The experiments show that INMON, and therefore interaction networks, can be used in character terminal interface and graphical user interface environments, and in analysis of both individual interactions and of sets of interactions. The interaction network was developed as a performance evaluation tool, and the experiments described show that interaction networks are very useful in performance evaluation. The fact that an interaction network is also very useful in providing a general understanding of the execution of a task, particularly when displayed, comes as a bonus.

The experiments were performed in the same environment as that described in Subsection 7.5.1 and Section A.1. Two Sun 3/50s were used, tui and weka. All user input occurred on devices connected to tui, with most processing performed on tui. Tui read most of its system programs from the NFS server weka, with all other files stored on

a local disc. The only processes on weka that took part in the tasks recorded in the experiments were `nfsd` processes, which service NFS requests.

All displays of interaction networks in this chapter were created by `inbrowser`, using the time layout.

11.1 Compiling a C Program

We now describe the analysis of two interactions, both of which involve the compilation and linking of a C program. The user input for both interactions was a newline character that was received by a shell process. In the first interaction, the newline was preceded by the input of '`cc hello.c`', and in the second by the input of '`cc -pipe hello.c`'. The `-pipe` option causes two of the processes that perform the compilation to communicate using a pipe, whereas ordinarily the two processes communicate using a temporary file.

The interaction with `-pipe` not specified is described in Subsection 11.1.1, and the interaction with `-pipe` specified is discussed in Subsection 11.1.2, together with a comparison of the two interactions.

11.1.1 A C Compilation

The task analysed in this subsection involves the compilation of a simple C program, hello.c, a program that prints the string "Hello world\n" to the terminal. The interaction network for the task is shown in Figure 11-1. It contains 1671 event vertices, considerably more than the 411 vertices in the "ls" interaction network of Figure 10-10. Fourteen processes are involved in the task: the shell process `csh`, five processes that perform the compilation (`cc`, `cpp`, `ccom`, `as`, and `ld`), and all eight `nfsd` processes on weka.

The `cc` process coordinates the compilation, and is responsible for managing the other four compilation processes. The `cpp` process is the C preprocessor. It acts on all of the preprocessor directives in the source file, with its main functions being to include ".h" header files, and to perform macro expansion. The `ccom` process is the C compiler. It takes the preprocessor output, and compiles it into assembly language. The `as` process is the assembler. It translates the assembly language statements into an object code file. The `ld` process is the link editor. It takes the object code file and, by linking in the C run time library, produces an executable program.

From the interaction network in Figure 11-1, it it clear that `cc` runs the four other compilation processes in sequence. There is little concurrency in the task, as indicated by the fact that all of the NFS requests are on the critical path.

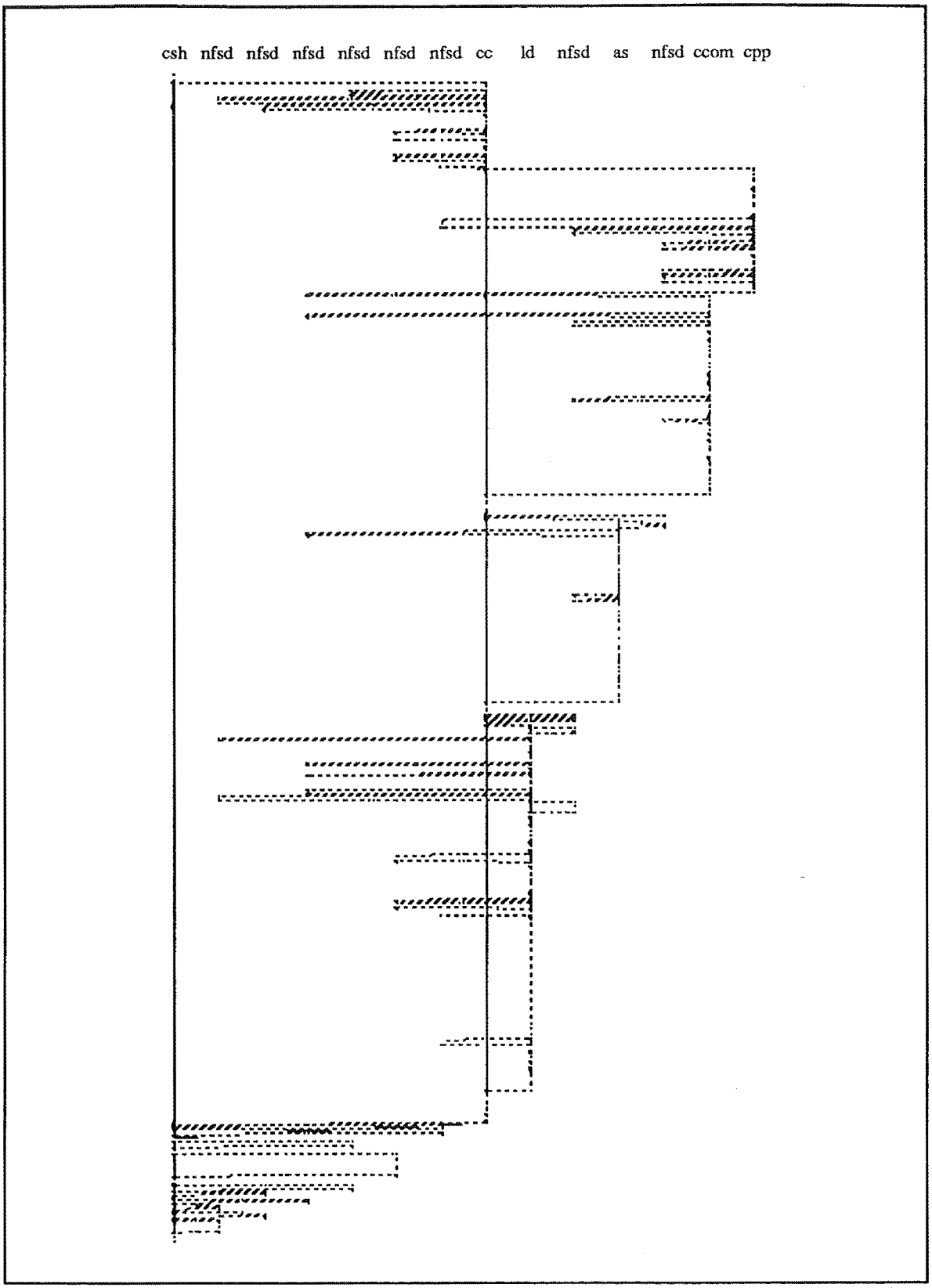


Figure 11-1: Interaction network for a C compilation

A summary is presented in Table 11-1 of some of the information produced by `analyse` from the interaction network. The format of Table 11-1 is used in the remainder of this chapter in presenting summaries of information produced by `analyse`. First reported in the summary is the response time of the interaction. Then, two decompositions of the critical path are presented. First, the largest of the state times are reported, followed by a decomposition into times spent on each node and on message passing between each pair of nodes. All times reported are in seconds. Finally, summaries of disc access counts and NFS remote procedure call counts are given. Two totals are given for each: a count of the number of accesses or calls on the critical path, and a count of all accesses or calls in the task. Then some decompositions are given of the count of all accesses or calls.

The large contribution made to response time by time spent in performing disc accesses is highlighted by the information in Table 11-1. Also, time spent in process execution on tui accounts for most of the length of the critical path.

Response time		13.04
CP decomposition by state		
run	4.31	
ready	1.12	
sleep_disc	4.50	
sleep_other	2.03	
queued	0.46	
network	0.69	
CP decomposition by node		
tui	10.45	
weka	1.54	
tui->weka	0.38	
weka->tui	0.72	
Disc accesses (on critical path / total)		70 / 70
Decomposition by access type		
read	46	
write	24	
Decomposition by node		
tui	35	
weka	35	
NFS RPCs (on critical path / total)		52 / 52
Decomposition by procedure called		
get attributes	12	
read symbolic link	6	
read data	31	
read directory	3	

Table 11-1: Summary of information produced by `analyse` for the C compilation interaction

11.1.2 A C Compilation With the `-pipe` Option

The task now discussed is compilation of the same program as discussed above, but with the `-pipe` option specified. When this option is specified, the `cc` process creates the `ccom` process straight after it has created the `cpp` process, and `cpp` communicates with `ccom` using a pipe rather than using a temporary file. The fact that the `cpp` and `ccom` processes run concurrently is very clear in the interaction network of Figure 11-2, which contains 1670 vertices. A consequence of this concurrency is that the critical path no longer includes all of the NFS remote procedure calls.

The `cpp` process writes all of its data to the pipe, then exits some time before `ccom` reads the data from the pipe. Two messages are shown as being sent from `cpp` to `ccom` through the pipe. The first message is 663 bytes of data, which is the entire input file after preprocessing. The second message is a close message.

The critical path runs through the first third or so of the execution of the `cpp` process, but then switches to the `ccom` process. The switch occurs in an `nfsd` process, which services a request for `ccom` immediately after servicing a request from `cpp`. An expanded view of the sub-graph of the interaction network where this switch occurs is shown in Figure 11-3. The vertical distances between Vertices v_1 to v_{10} are directly proportional to time. Vertices v_{11} to v_{15}, which relate to a disc access, have been omitted. From the figure, we can see that `cpp` and `ccom` make NFS requests to weka in quick succession, with the `ccom` request made about 12ms after the `cpp` request. When the NFS request from `cpp` arrives at weka, all eight `nfsd` processes are awoken. Before the `cpp` request can be serviced, the `ccom` request arrives, and joins the request queue. An `nfsd` process is then resumed, and performs the `cpp` request. No disc accesses are required, so the reply is sent almost immediately. The same `nfsd` process then receives the `ccom` request, and services it.

Now consider the critical path through `cpp` and `ccom`, taking vertices in reverse order of time, as is done by the algorithm that computes the critical path. The critical path goes through the last vertex of `ccom`, and from there travels back through `ccom`. Two join events are encountered, but in each case the joining message from `cpp` is waiting (and has been waiting for a long time) so the critical path remains with `ccom`. The critical path remains mostly with ccom (with the exception of the durations of some NFS requests) for about three quarters of the execution of `ccom`. Then, the critical path travels back along an NFS reply to Vertex v_{16} of Figure 11-3, and then back through the servicing of the request to Vertex v_9. At Vertex v_9 the critical path algorithm has to choose which edge the critical path will follow. The joining message has been queued for some time, whereas the `nfsd`

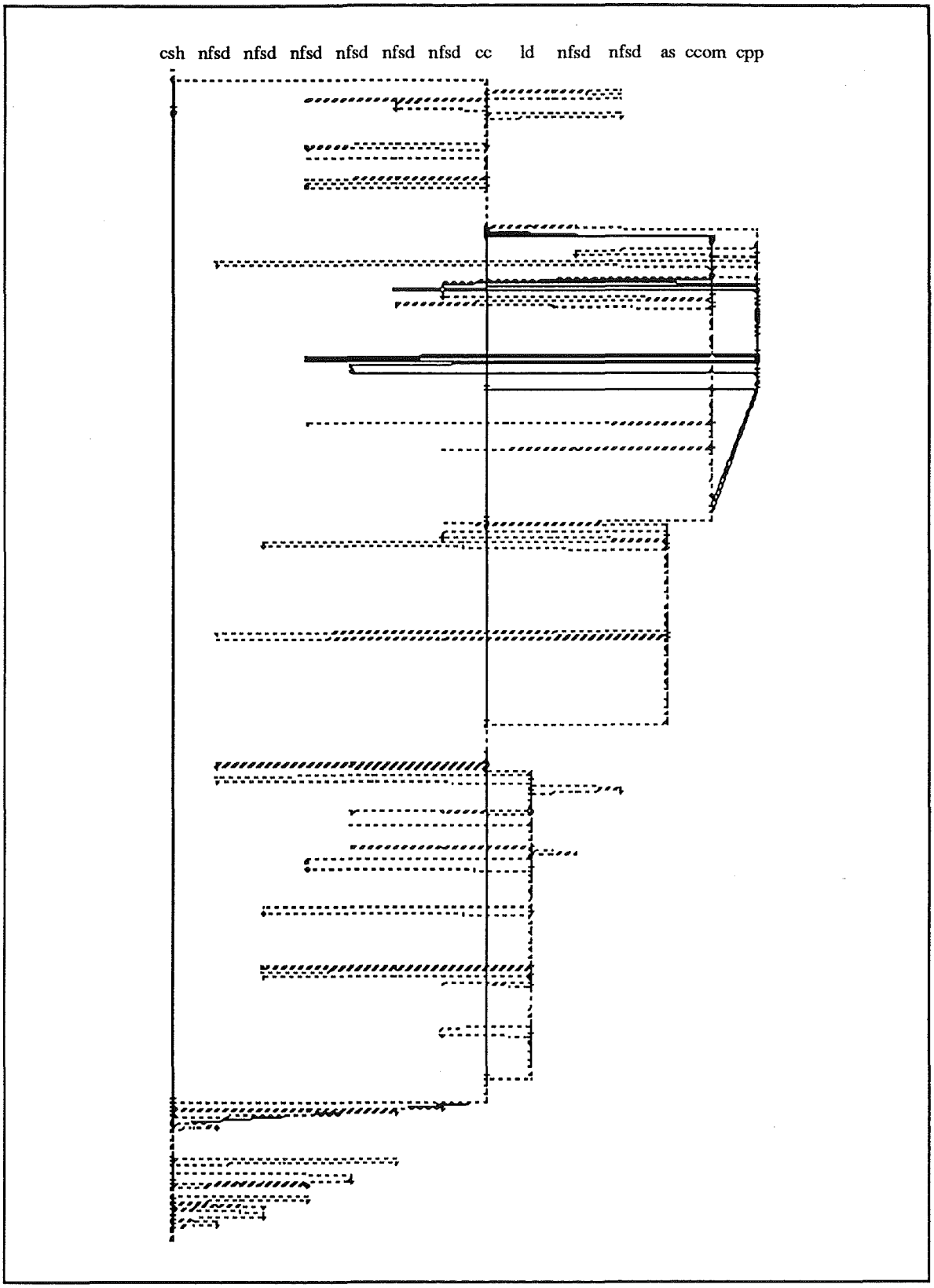


Figure 11-2: Interaction network for a C compilation: -pipe option

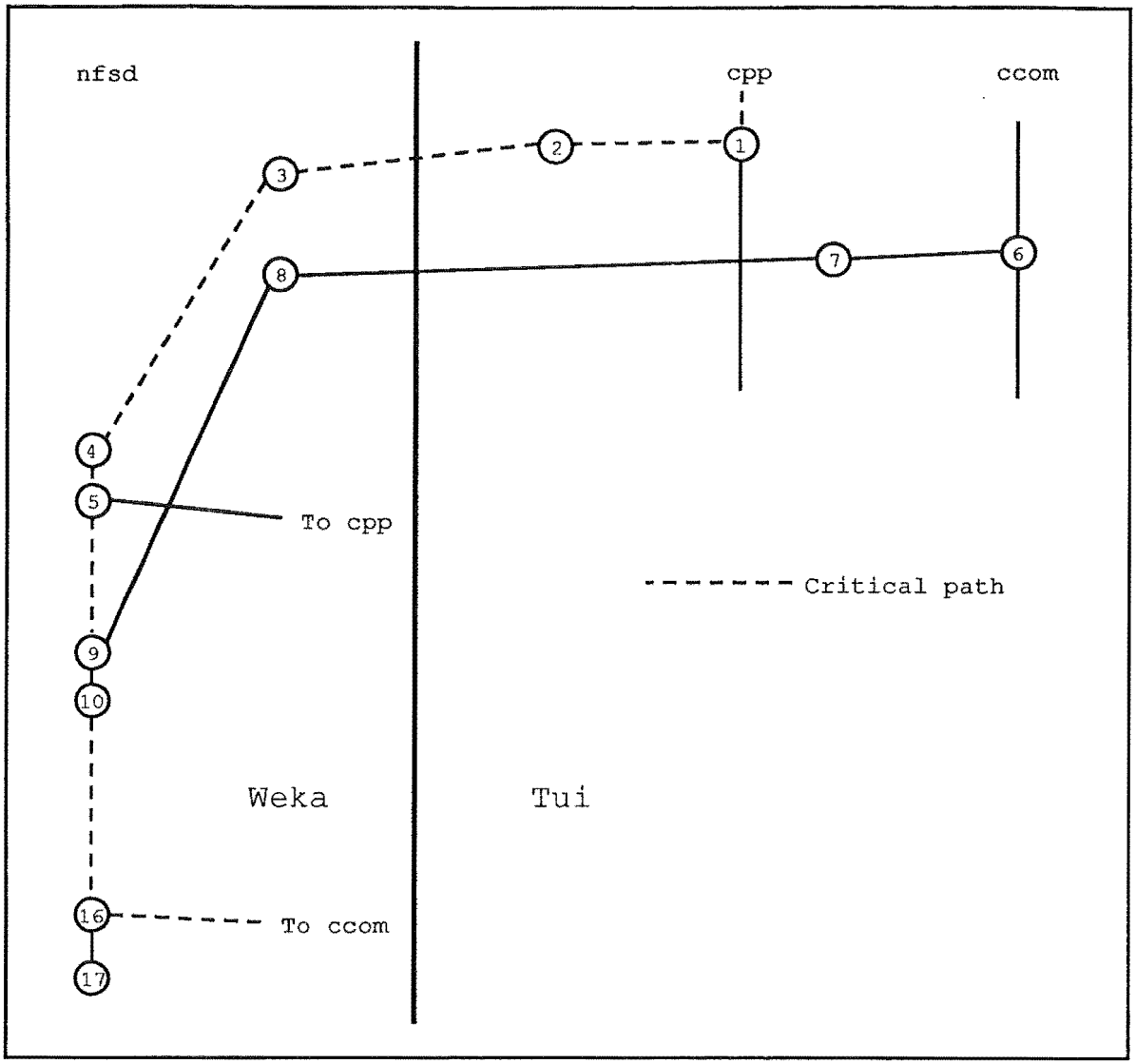


Figure 11-3: The switching of the critical path from `cpp` to `ccom`

process has only just finished its previous request, so the critical path stays with the `nfsd` process, and travels back through an NFS request to the `cpp` process. From that point the critical path remains with the `cpp` process, except for the periods of some NFS requests, back to the point at which the `cpp` process was created.

This is an example where the critical path has transferred from one process to another because of the use of a shared resource, in this case a file server process. See part (2) of Subsection 10.4.1 and Section 11.3 for further discussion.

A summary is presented in Table 11-2 of some of the information produced by `analyse` from the interaction network. The occurrence of some concurrent processing is clear, as not all disc accesses and NFS remote procedure calls are on the critical path.

Also, the response time for this interaction is 1.67s shorter than the response time of the task described in Subsection 11.1.1. A possible explanation is that the response time for the compile where the `-pipe` option was used is shorter because of the concurrency between the `cpp` and the `ccom` processes.

To check this possibility, the elapsed time from the beginning of the interaction to the time at which the `ccom` process terminated was determined for both interactions. Where the `-pipe` option was not used, the `ccom` process terminated after 4.74s, and where the `-pipe` option was used, the `ccom` process terminated after 4.42s, a difference of 0.32s. So the extra concurrency could account for some of the improvement in the response time, but by no means all. The major difference in the components of response time for the two interactions, is the time spent in the `sleep_disc` state: 4.50s as against 2.74s. The number of disc accesses on the critical path is, however, much the same; 70 as against 62. Most of the reduction in the `sleep_disc` state most likely arises from less time spent queued for disc access.

Response time		11.37
CP decomposition by state		
run	3.89	
ready	1.06	
sleep_disc	2.74	
sleep_other	2.49	
queued	0.44	
network	0.72	
CP decomposition by node		
tui	8.82	
weka	1.38	
tui->weka	0.43	
weka->tui	0.74	
Disc accesses (on critical path / total)		62 / 66
Decomposition by access type		
read	46	
write	20	
Decomposition by node		
tui	32	
weka	34	
NFS RPCs (on critical path / total)		46 / 52
Decomposition by procedure called		
get attributes	12	
read symbolic link	6	
read data	31	
read directory	3	

Table 11-2: Summary of information produced by `analyse` for the C compilation interaction with the `-pipe` option

11.2 Pipelines

We now describe the analysis of two interactions which involve a process "pipeline", that is one process sending data to another through a pipe. The user input to start each interaction was a newline which was received by a shell process. In both interactions, the newline was preceded by input of 'more file | wc', where file contains nearly 250Kb, and is stored on weka. The more command simply copies the file to the pipe, and the wc command prints counts of the number of lines, words, and characters it reads from the pipe. The difference between the environments in which the two interactions were executed is that, in the second, the readahead function was disabled. The readahead function, when enabled, attempts to bring into memory each block of a file being read sequentially shortly before the block is needed by the reading process.

The pipeline executed with readahead enabled is described in Subsection 11.2.1, and the pipeline executed with readahead disabled is discussed in Subsection 11.2.2, together with a comparison of the two tasks.

11.2.1 A Pipeline With Readahead Enabled

The interaction network for the execution of the pipeline with readahead enabled is shown in Figure 11-4, and contains 1599 vertices. Seven processes were involved: the shell process csh, the more and wc processes of the pipeline, and four of the nfsd processes on weka. The shell process created the more process first, and shortly after created the wc process. A record of the more than fifty messages passed from more to wc through the pipe can be seen in the interaction network. The more process made an NFS request immediately before sending its first message to wc. Thereafter, the more process made no further NFS requests, despite the fact that it was reading a 250Kb file. This shows the effectiveness of readahead for this task.

There is considerable concurrent operation of the more and the wc processes. During the period that these two processes exist, the critical path is in the more process for the most part, although it switches into the wc process on four occasions. On each occasion the critical path accompanies a data message en route to wc, and accompanies a backflow message when returning to more. Other backflow messages are not on the critical path. During the second period in which the critical path has switched to the wc process, there are several consecutive <data message, backflow message> pairs. More information on backflow messages is given in Subsection 9.2.3.

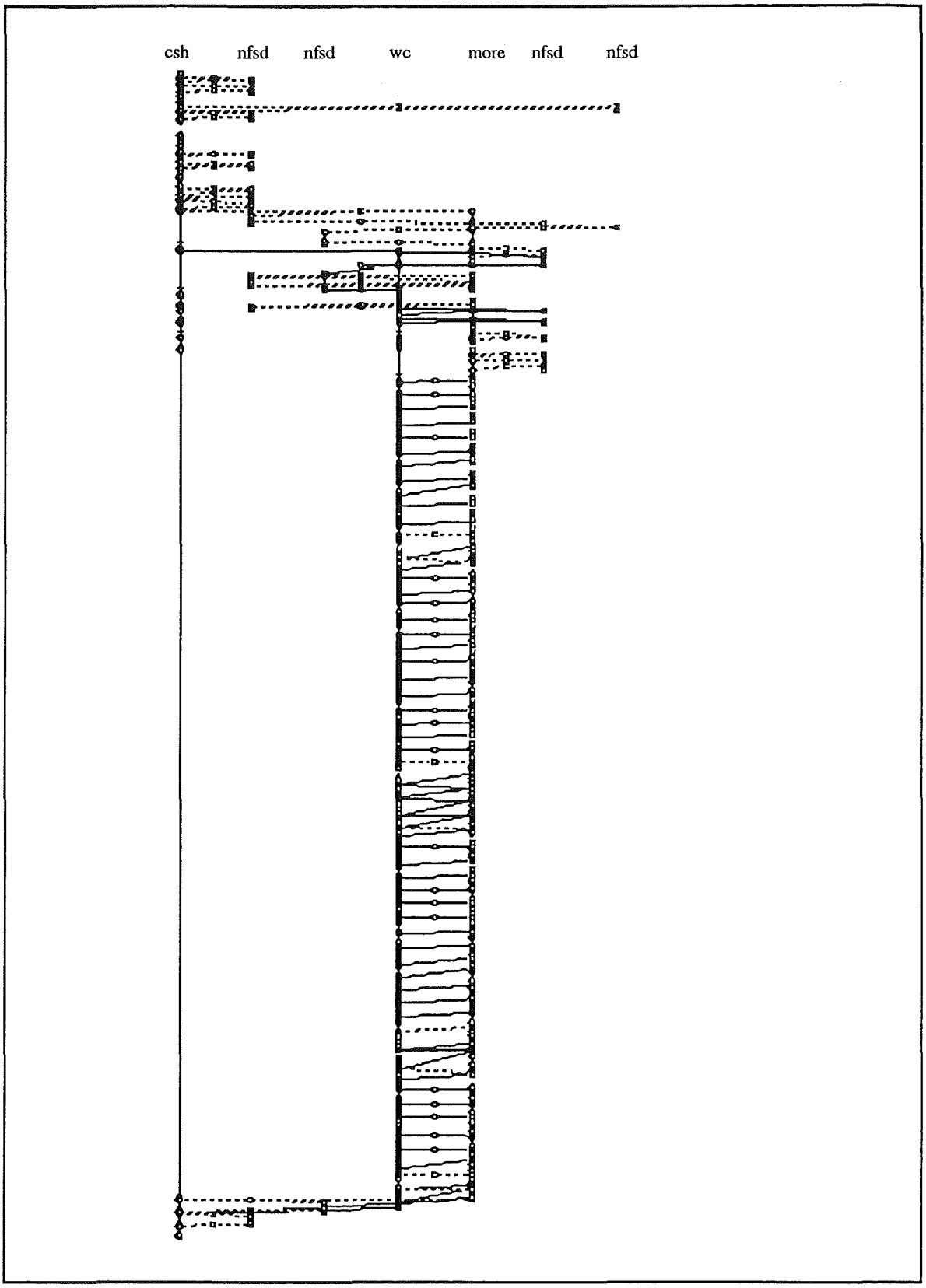


Figure 11-4: Interaction network which includes a pipeline

A summary is presented in Table 11-3 of some of the information produced by `analyse` from the interaction network. The relatively large amount of time spent in the `ready` state is probably caused by competition for the processor between the `wc` and `more` processes.

Response time		9.17
CP decomposition by state		
run		3.22
ready		4.34
sleep_disc		0.78
sleep_other		0.15
queued		0.26
network		0.42
CP decomposition by node and process		
tui		8.82
more	6.03	
wc	1.13	
shell	0.88	
weka		1.38
tui->weka		0.43
weka->tui		0.74
Disc accesses (on critical path / total)		21 / 23
Decomposition by node		
tui		11
weka		12
NFS RPCs (on critical path / total)		23 / 25
Decomposition by procedure called		
get attributes		2
directory lookup		1
read symbolic link		2
read data		20

Table 11-3: Summary of information produced by `analyse` for the pipeline interaction; readahead enabled

11.2.2 A Pipeline With Readahead Disabled

The interaction network for the execution of the pipeline with readahead disabled is shown in Figure 11-5, and contains 2142 vertices. This task proceeds in much the same way as the task with readahead enabled, the most significant difference being that many NFS requests are made by `more` as it is reading `file`, whereas with readahead enabled very few NFS requests are made by `more` to read `file`. Because of the greater number of NFS requests, this task involved all eight `nfsd` processes on weka.

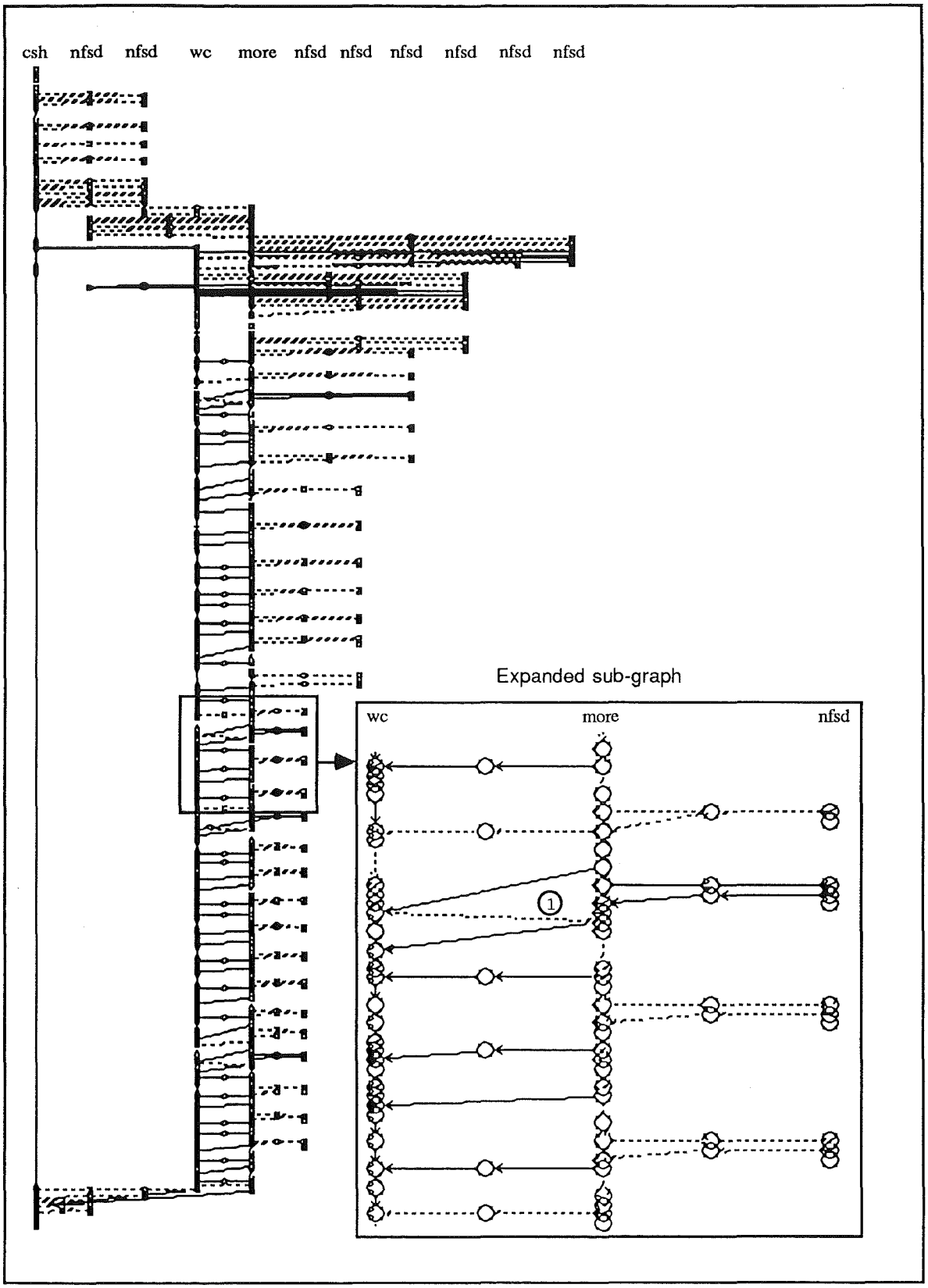


Figure 11-5: Interaction network which includes a pipeline; readahead disabled

It is apparent from Figure 11-5 that `more` makes one NFS request for every two messages written to the pipe, and this is shown particularly clearly in the inset, which is a blow-up of a sub-graph of the interaction network. One NFS request is made for every two messages written to the pipe because an NFS read request returns up to 8192 bytes of data, whereas the capacity of a pipe is 4096 bytes. The inset also clearly shows the critical path going from `more` to `wc`, then returning to `more` along the backflow message labelled with the number 1.

A summary is presented in Table 11-4 of some of the information produced by `analyse` from the interaction network. The readahead function is normally enabled, and the effects of disabling it are clear, both from the interaction network in Figure 11-5, in which the extra NFS requests required are clearly shown, and in the statistics computed by `analyse`. The number of disc accesses on weka, the NFS server, increased from 12 to 41. The number of NFS remote procedure calls increased from 25 to 63. The time spent in message `queueing` and `network` states increased from 0.68s to 1.72s.

Response time		10.33
CP decomposition by state		
run		3.68
ready		4.01
sleep_disc		0.83
sleep_other		0.09
queued		0.59
network		1.13
CP decomposition by node and process		
tui		7.97
more	6.08	
wc	1.12	
shell	0.77	
weka		0.65
tui->weka		0.45
weka->tui		1.19
Disc accesses (on critical path / total)		50 / 55
Decomposition by node		
tui		14
weka		41
NFS RPCs (on critical path / total)		53 / 63
Decomposition by procedure called		
directory lookup		11
read symbolic link		2
read data		50

Table 11-4: Summary of information produced by `analyse` for the pipeline interaction; readahead disabled

Interestingly, the time spent in `sleep_disc` state increased only slightly (0.78s to 0.83s), despite the fact that the number of disc accesses on the critical path more than doubled. Without a decomposition of the `sleep_disc` times into time spent queued for disc access and time spent performing disc accesses, it is difficult to say why this might be. It is possible that the 0.78s includes a much greater queueing component than the 0.83s. Also, most of the additional accesses are to a single file (`file`), and as SunOS attempts to keep all of the disc blocks for one file close to each other, the seek times for the additional accesses may be small.

11.3 Mouse Input

We now analyse an interaction network recorded for an interaction with a graphical user interface. The interaction was recorded in an X windows environment [SCHE90]. A client/server approach is used in X. An X server process is associated with each display, and with the set of input devices, typically a mouse and a keyboard, associated with that display. The X server is the only process that interacts directly with the devices, and it is responsible for managing the display, and for receiving all user inputs from the input devices.

Applications, known as clients, use the X server as a provider of display and input services. Each client can request that the server establish windows. The client can output to a window, which causes the server to update the contents of the window on the screen. Also, user input actions performed when a window is the one currently active, are relayed to the client associated with the window. A special class of client programs are the window managers, which control the layout of windows on the screen, and which usually provide the user with the ability to move windows, to resize windows, and to select which of a group of overlapping windows should be displayed in the foreground. Up to one window manager can be associated with a display.

The interaction now considered occurred as part of a window resize operation. The window manager in use was the `tvtwm` window manager. A window managed by `tvtwm` is resized if the user:

(1) moves the mouse cursor to the resize icon in the title bar of the window which is to be resized,

(2) presses the left mouse button,

(3) moves the mouse cursor over the window edge to resize, then moves the cursor to the new position of the edge, and

(4) releases the left mouse button.

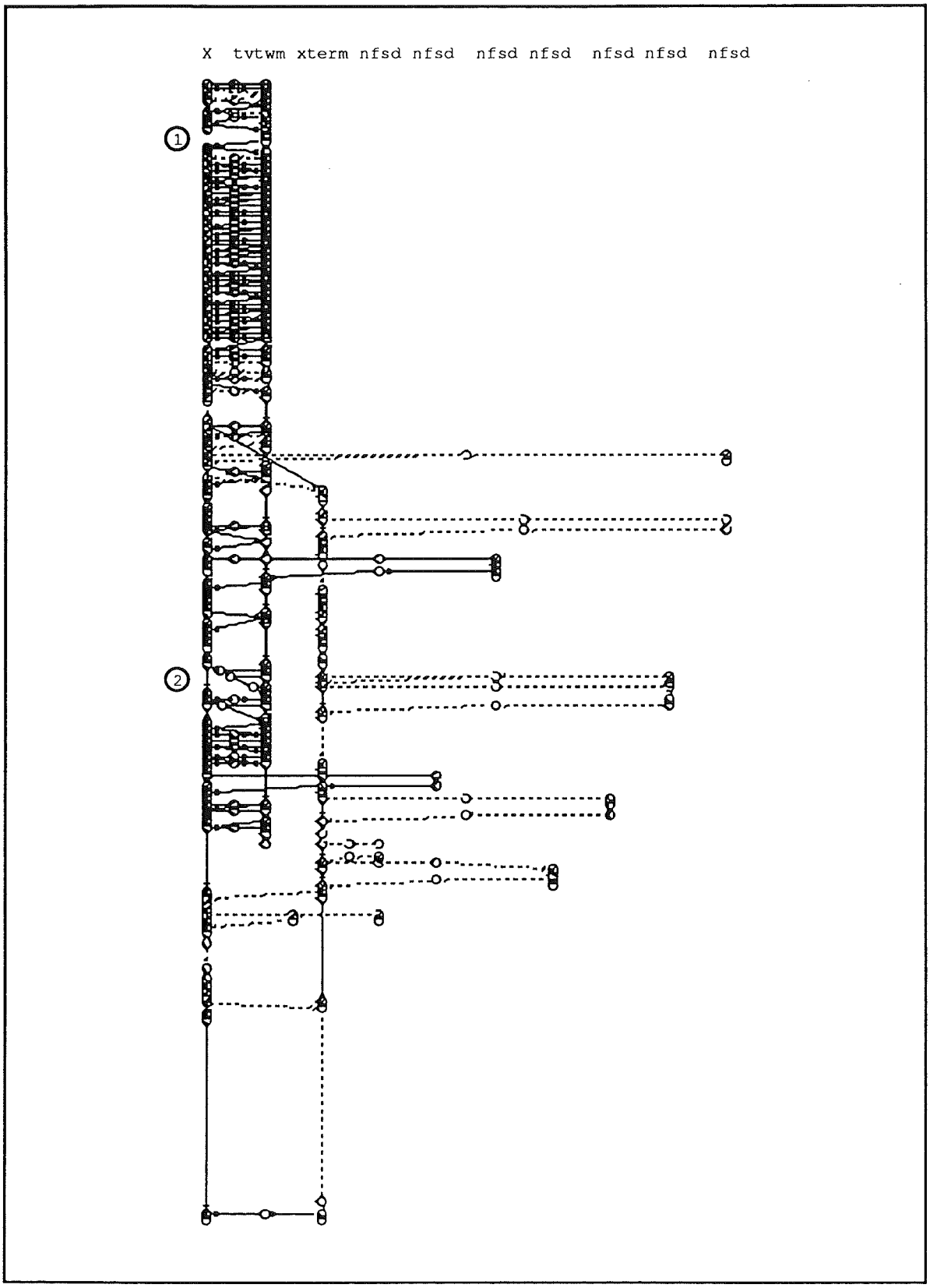


Figure 11-6: Interaction network for the window resize task

The user action of the interaction now considered is action (4) above, that is the release of the left mouse button at the end of a window resize operation. The window width was increased by the resize operation. The interaction network of the window resize task appears as Figure 11-6, which contains 1326 vertices. Ten processes participate in the task. The three processes on tui are: x, the X server process, `tvtwm`, the window manager, and `xterm`, the client whose window is resized. The seven processes on weka are all `nfsd` processes.

The user input from the mouse is received by the x process. For the first part of the task, many messages are exchanged between the x and `tvtwm` processes. The x process then sends two messages to the `xterm` process, informing it that its window size has changed. After this, further messages are exchanged by the x and `tvtwm` processes, but at a slower rate than before, probably because these two processes have to share the processor with `xterm`. The `xterm` process communicates a little with the x process. All communication is between the server and clients; a window manager client cannot communicate directly with another client.

Messages between x and `tvtwm` alternate in direction with the first and the last from x to `tvtwm`. The alternation is difficult to see from the figure, but can be readily seen by using the zoom feature of either `Xgrab` or `inbrowser`. The critical path runs through the x process for most of the period of concentrated communication. One would have expected, however, that the critical path would zigzag back and forth between the two processes, because after sending a message each process must wait for a reply from the other process. The critical path remains mainly within the x process, because it contains few B-edges during the period of heavy communication.

The reason that there are so few B-edges is not obvious, but what seems to be happening is the following. Let us assume that: the critical path is currently in the x process, the x process is the one currently executing, and the `tvtwm` process is asleep, awaiting a message. The x process sends a message to `tvtwm`, which causes `tvtwm` to be awoken, and because `tvtwm` has a higher priority then x (because `tvtwm` has used less processor time recently) the x process is put back onto the processor queue, and the `tvtwm` process is executed. The `tvtwm` process receives the message, sends a reply, then goes back to sleep waiting for the next message. This allows the x process to continue. It tries to receive a message, and finds a message waiting for it, so does not need to block waiting for the message.

During the execution of `tvtwm`, the x process is blocked waiting for the processor, rather than a message, so it is assumed that the critical path goes through the x process. The critical path should have zigzagged between x and `tvtwm` because of the contention for

a shared resource, in this case the processor. See part (2) of Subsection 10.4.1 for further discussion.

An observation that can be made from the interaction network is that a lot of context switching occurs between the X and `tvtwm` processes during the period of heavy communication.

During the interaction, four other user inputs are received and processed briefly by the X process. The tasks for each of the other user inputs are recorded in separate interaction networks. There are, therefore, four breaks in the event chain of the X process. One break, labelled with the number 1, is easily seen on Figure 11-6. During each break the X process is performing work for some other task, and the critical path switches from X to `tvtwm`. Then, after the break, the critical path switches back to X again.

The communication between X and `tvtwm`, and between X and `xterm`, was through Unix domain stream sockets. Although no message splitting and joining was recorded for most of the PE_DEQUEUE events, message splitting did occur on a few occasions. There are four instances of message splitting, where the X process sent a message of length 32N bytes (N > 1), and `tvtwm` received the message 32 bytes at a time. The values of N are 2, 3, 7, and 2. One of these instances, labelled with the number 2, can be seen on Figure 11-6.

There is also one instance of message joining. The two messages sent by X to `xterm`, at the beginning of the participation of `xterm` in the task, are received by `xterm` in a single operation (although this is not clear from Figure 11-6). The critical path goes through this PE_DEQUEUE message join/split vertex, with the incident edge selected by the critical path algorithm being the edge associated with the message that arrived most recently, as discussed in Subsection 10.4.1. This is clear from the diagram.

In this interaction network, final vertex v_f is not in the same process as the source vertex v_0. Also, the interaction network for a window resize of an `xterm` (terminal emulator) process, may extend to the processes being run from the `xterm`. This is because `xterm` sends a change window signal (SIGWINCH) to all processes being run from the `xterm` process. Any processing done by these processes in response to handling the signal should also be included in the interaction network, but currently no probes record the sending and receiving of signals.

A summary is presented in Table 11-5 of some of the information produced by `analyse` from the interaction network. As noted above, the contribution of `tvtwm` to the critical path is probably underestimated.

The recording analysis of the window resize task shows clearly the large amount of processing done for an apparently trivial operation.

```
Response time                                                    3.92
   CP decomposition by state
      run                                              0.67
      ready                                            1.13
      sleep_disc                                       1.09
      sleep_other                                      0.67
      queued                                           0.21
      network                                          0.15
   CP decomposition by node and process
      tui                                              3.32
            X                                  1.34
            tvtwm                              0.25
            xterm                              1.73
      weka                                             0.24
      tui->weka                                        0.04
      weka->tui                                        0.19
      tui->tui                                         0.13

Disc accesses (on critical path / total)                    23 / 25
  Decomposition by node
      tui                                            18
      weka                                            7

NFS RPCs (on critical path / total)                          8 / 10
   Decomposition by procedure called
      read data                                      10
```

Table 11-5: Summary of information produced by `analyse` for the window resize task

11.4 A Set of Interactions

The last experiment to be described is one in which the processing for a set of interactions, rather than a single interaction, is analysed. While the ability to study individual interactions is important, a major aim in performance analysis is to evaluate performance for sets of interactions. Examples of sets of interactions that a performance analyst may wish to evaluate include: all interactions that occur during some period; all interactions that involve use of some piece of hardware, such as a file server node; or all interactions that make use of some piece of software, such as a database management system.

The set of interactions to be analysed in this experiment had to be selected. As in the clock synchronisation experiments described in Appendix A, weka and tui were under light workloads at the time of the experiment, so some form of artificial workload had to be used. The interactive workload of the MUSBUS version 5.0 benchmarking package [MCDO87] was selected to provide the interactions to monitor. The operation of the interactive benchmark of MUSBUS is described in Subsection 11.4.1, with changes made

to facilitate monitoring described in Subsection 11.4.2. The experiment and its results are discussed in Subsection 11.4.3.

11.4.1 MUSBUS Interactive Benchmark

The MUSBUS interactive benchmark allows the inputs of one or more users to be simulated, with the characters input by each simulated user being read from a specified script file. For the remainder of this section, we shall refer to "a user" rather than to a "simulated user". Four script files are provided with MUSBUS, each of which consists of a set of inputs for the command interpreter `sh`. The same set of commands appears in each script, but with small variations in their ordering. Each script contains sixteen command lines, with nine different commands used. The `mkdir` (create directory), `grep` (search file for pattern), `ls` (list directory), and `cat` (display file) commands appear once; the `cc` (compile and link C program), `cp` (copy), `chmod` (change file protection), and `ed` (edit file) commands appear twice; and the `rm` (delete file) command appears four times.

Of these nine commands, only `ed` requires additional input. There are two different scripts of `ed` commands, each used once in each of the four `sh` scripts.

Benchmark execution is controlled by the `makework` program, which has as a parameter `nusers`, the number of users which it is to simulate. The `makework` program reads an input file that contains one line per user, with each line containing a number of pieces of information about a user, in particular the name of the program to which user input should be directed, and the name of the script file that contains the user input. The `makework` program creates a pipe and a process for each user. The process is setup to take its standard input from the pipe, and to execute the program specified for that user.

The `makework` program then, at intervals, writes parts of each script to the appropriate pipe, with some effort being made to simulate user typing speeds. For each of its child processes, `makework` determines the number of characters to write to the pipe connected to the child process, by generating a random number from a uniform distribution in the range 1 to 60. A pointer associated with the script is advanced by the number of characters determined, and any input lines past which the pointer advances are written to the pipe, with each line written by a single `write` system call.

The `makework` program uses two mechanisms to simulate user typing. First, script file characters are written to each pipe one line at a time, simulating the line buffering provided by Unix terminal device drivers. Second, the rate at which script characters are written to pipes is governed by the `rate` parameter, an estimate of the user typing speed in characters per second. `rate` influences the intervals between consecutive passes which write to the pipes. The simulation of user typing is somewhat unrealistic, in that a user is assumed to continue to type input regardless of whether previous commands have finished.

11.4.2 Instrumenting `makework`

The execution of the `makework` program is a single interaction, which is associated with the user action of typing the newline at the end of the `makework` command line. Each time `makework` writes a line of a script file to a pipe, however, it is simulating a user input, and we would like to have INMON treat the simulated user input as a real user input. To do this required a probe to be installed in `makework`, and a new system call, `getnextstn`, which provides access to the kernel function that generates new sub-task numbers (see Subsection 9.3.1).

The instrumentation of `makework` is now described. Most of the activities of the `makework` process are not of interest, so the `setstn` system call is used at the start of the program to set the sub-task number of the `makework` process to PE_INVALIDIPID. When a line is about to be written to a pipe, a new sub-task number is generated and assigned to the `makework` process. A TERM_INPUT_END event record is created to record the "input" of the line, and is recorded using the `perf` system call described in Section 9.4. The line is then written to the pipe, causing a SOCK_SEND event to be recorded which relates the current sub-task number of the `makework` process to the new sub-task number assigned to the line of data written to the pipe. The sub-task number of the `makework` process is then reset to PE_INVALIDIPID using the `setstn` system call. The message in the pipe is read at some later time, and the rest of the task is linked to the original write to the pipe, by the sub-task number associated with the message in the pipe.

The time at which the line is written to the pipe is the time at which the simulated user typed the data, and the time that the line spends in the pipe corresponds to the time spent by the data in the terminal input buffer.

Also, some changes had to be made due to the way in which simulated user input was provided for the executions of the `ed` command. In the original MUSBUS scripts, user input was supplied to the `ed` process by another process through a pipe. Because `makework` had been instrumented to record simulated user inputs, it was desirable that `makework` supply all input from a single script file, rather than just the `sh` input.

The script files were modified to include all of the `ed` input, but a synchronisation problem, described in [MCDO87], then had to be solved. Consider the following sequence of events relating to one <script file, pipe, `sh` process> triple. Assume that initially the pipe is empty, and `sh` is performing the last command that it read from the pipe. The `makework` process then writes two further lines from the script file to the pipe. The first is a command intended for `sh`, instructing it to perform the `ed` command, and the second is a command intended for `ed`. When the shell next reads from the pipe, it is prepared to read up to 128 characters, and so reads both lines from the pipe. Because of this, `ed` never receives the

command that was intended for it, and after `ed` finishes `sh` will attempt to execute the command intended for `ed`.

To solve this problem, `makework` was modified so that whenever a ! appeared in a script file, that line of the script file was not written to the pipe until the pipe was empty. Also, ! characters were inserted in each script file wherever the input destination changed.

11.4.3 The Experiment

An experiment was performed whereby `makework` simulated 4 users, and where the scripts used were the four standard MUSBUS scripts. The value used for `rate` was the default value of 5 characters per second. With the addition of the `ed` commands, each script file contained 54 command lines, so there were 216 interactions in the set analysed. The 216 task log files contained a total of 30839 event records.

Before analysing the set of interactions, we examine the execution of one of the largest interactions in the set, as its analysis highlights some important points about the set of interactions. The interaction network for this interaction, shown in Figure 11-7, contains 1640 vertices. The interaction represents a `sh` process reading several command lines in a single read, and then executing the specified commands. Five command lines are read, which execute the commands `grep`, `cat`, `cc`, `rm`, and `ed` respectively. The `cc` command in the interaction network compiles only to object code, so `ld` is not executed.

The execution of several commands by a single task was observed a number of times in the set of interactions. The causes were: first, the fast simulated typing rate and the fact that command lines were written to pipes regardless of whether previous commands had finished (except for the synchronisation points described in Subsection 11.4.2); and second, the fact that `sh` reads up to 128 characters at a time, so it is possible for it to read several commands from the pipe in a single read. The user input for the interaction we are currently discussing was the command line of the `ed` command, the last of the commands executed. The other commands originated in other tasks, but were joined with this task when `sh` read all of the commands from the pipe in a single read. The interaction networks for these other tasks contain few events, basically just the write of the command line to the pipe.

From the interaction network, we can see that the characters of the `ed` command spend a considerable amount of time in the pipe. Also, the interaction network includes only part of the execution of `ed`, as `ed` receives another user input, and so becomes involved in a different task (in fact `ed` receives many inputs, and is involved in many tasks). This again highlights the difference between program and interaction analysis.

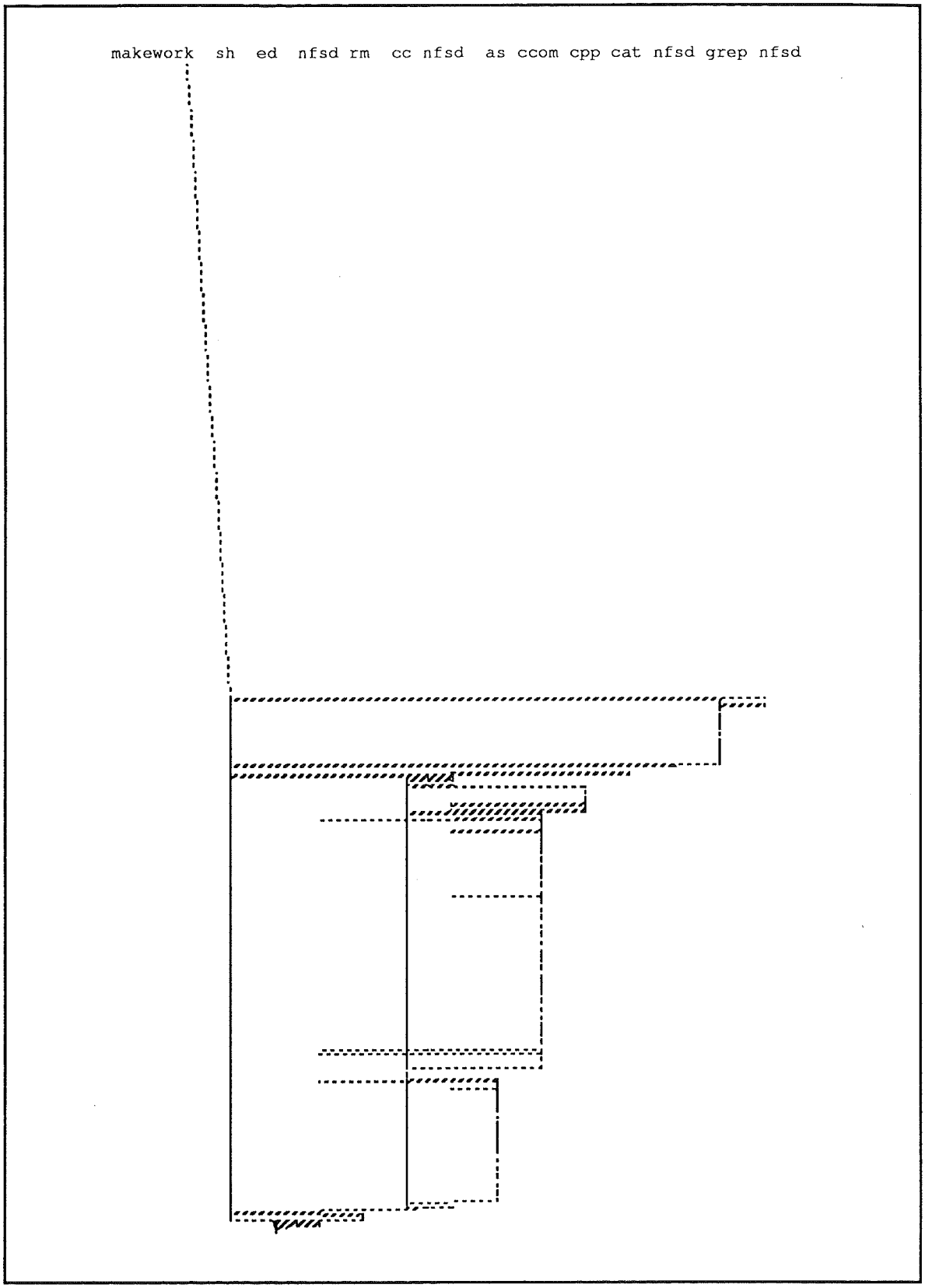


Figure 11-7: Interaction network for an interaction in the MUSBUS set, which shows execution of multiple commands in a single task.

A summary is presented in Table 11-6 of some of the information produced by `analyse` from the set of 216 interaction networks.

Response time		4132.87
CP decomposition by state		
run	84.47	
ready	341.33	
sleep_disc	82.80	
sleep_other	35.33	
queued	3581.15	
network	5.19	
CP decomposition by node and process		
tui	542.77	
weka	3.78	
tui->weka	3.30	
weka->tui	7.89	
tui->tui	3575.15	
Disc accesses (on critical path / total)		901 / 903
Decomposition by access type		
read	433	
write	470	
Decomposition by node		
tui	856	
weka	47	
NFS RPCs (on critical path / total)		464 / 466
Decomposition by procedure called		
get attributes	41	
directory lookup	53	
read symbolic link	164	
read data	196	
read directory	12	

Table 11-6: Summary of information produced by `analyse` for the MUSBUS set of interactions

By far the largest component of the lengths of the critical paths is the time that messages spent queued en route from one process on tui to another. This is the time that input lines waited in pipes with, for example, more than half of the critical path in Figure 11-7 being associated with a message waiting in a pipe. In practice, users type very few characters while they are waiting for a previous command to complete, so this very large critical path component can be ignored, as it has arisen because of the unrealistic way in which user input is simulated in MUSBUS. Analysis using INMON has highlighted this shortcoming of MUSBUS.

Of the remaining components, the major bottleneck is the processor on tui, with 423.45s of critical path time spent using or waiting for tui's processor. This represents

76% of the sum of the lengths of critical paths, after the exclusion of the times that input lines spent in pipes.

The fact that nearly all disc accesses and NFS requests are on a critical path indicates that very little concurrent activity has occurred within individual interactions of the set. Also, it is apparent that times spent in making NFS requests to weka have little effect on response times. With further analysis, the contributions could be determined of each command and command type to total critical path time.

11.5 Summary

In this chapter, several experiments have been described, which involved the recording and analysis of interaction networks by INMON. The main aims in conducting these experiments were to show that interaction networks could be used in analysis of single interactions and of sets of interactions, and to show that interaction networks could be used in graphical user interface environments. We consider that the use of interaction networks in analysis of sets of interactions to be at least as important as their use in analysis of individual interactions, even though the fact that three of the four experiments involved the analysis of individual interactions might indicate that this is not the case.

In the first three experiments, individual interactions were analysed. For the first two experiments, user input was from a keyboard, and the user interface was a character terminal interface. In the third, input was from a mouse, and the user interface was a graphical user interface. In the fourth experiment a set of interactions was analysed.

In the first experiment, compilation of a C program was performed, both with and without the `-pipe` option, was investigated. In the second experiment, the execution of a pipeline of two processes, and the impact of the readahead function, were investigated. The concurrent execution of the two processes that communicated through the pipe was clear from the interaction networks presented for this experiment.

In the third experiment a single interaction, which resulted from a "mouse button release" user input, was analysed. This experiment highlighted the differences between an analysis from the interaction viewpoint, as taken by this work, and an analysis from the program viewpoint used in most other approaches. Where a graphical user interface is used, rapid response to many types of user input is very important, as slow responses for trivial commands cause high user dissatisfaction. For this reason it is important to be able to isolate and analyse the processing performed as a result of each user input, as opposed to analysing the entire execution of one or more programs. An interesting observation from the experiment was that a surprisingly large amount of processing occurred as the result of an input that completed a window resize operation.

In the final experiment, a set of 216 interactions was analysed. In this experiment user input was simulated, with a coordinator process feeding "user input" to other processes through pipes. This experiment showed that INMON can be used for the analysis of sets of interactions, as well as for the analysis of individual interactions. The bottleneck for this set of interactions was identified as being tui's processor. Also, detailed breakdowns are available for disc accesses and NFS requests, and for individual commands and command types.

The experiments described in this chapter show that INMON, and therefore interaction networks, provide valuable performance information in a wide variety of situations. In particular, the experiments show that the interaction network concept is well suited for use in performance evaluation of graphical user interfaces, and that INMON can be applied to sets of interactions as well as to individual interactions. Also, the information available from an interaction network is sufficiently detailed and presented in such a form, that a programmer should find interaction networks a very useful tool in enhancing understanding of a system's behaviour, and therefore particularly useful in debugging. Finally, the experiments indicate that the assumptions made about continuing sub-tasks and task boundaries (as described in part (6) of Subsection 4.2.4) are valid.

Chapter 12

Conclusions

The most important concept introduced in this thesis is the *interaction network*, a way of describing the execution of a *task*, where a task is **all** of the processing resulting from a single user input to a loosely-coupled distributed system. Several methods have been described in the thesis to show how performance information can be extracted from interaction networks. The author considers that the interaction network is a very powerful tool for evaluation of interactive performance of loosely-coupled distributed systems.

A performance monitor based on the interaction network concept has been successfully implemented, and has been found to provide both useful performance information and valuable insights into other aspects of task execution.

At the start of the thesis, the author described a model for computation in distributed systems, which is used as the basis for the definition of the computational environment for tasks. The model is simple, defining a distributed computation to consist of communicating threads, and general, applying to many different hardware architectures (centralised systems, and tightly-coupled and loosely-coupled distributed systems) and software architectures (centralised, network and distributed operating systems, for instance).

The author has introduced a model for user interaction, which defines task boundaries. In this model, a user is seen as performing a series of *user actions*, each of which is a single input. A task is all of the processing performed in response to a single user action. This model of user interaction covers use of a wide range of user interfaces, from command-line user interfaces to graphical user interfaces. Analysis of *interactions* is very different to analysis of programs, the most widely used approach to performance analysis of distributed systems. Use of the interaction as the unit of analysis is an important contribution of this work.

An interaction network is a directed acyclic graph, which describes the execution of a task. Each vertex represents an event, such as the creation of a thread, or a thread sending or receiving a message. Each edge represents some activity associated with a thread or with a message. A *critical path* through an interaction network is a sequence of edges such that to reduce response time for an interaction, the length of at least one of the edges in the

sequence must be reduced. The author has introduced a simple algorithm for determining the critical path through an interaction network.

Several methods for deriving performance information from interaction networks have been described. One important analysis method is the decomposition of a critical path into times spent in different states, where each state is defined by use of some combination of system objects. Decompositions can be computed at different levels of aggregation. Another important analysis method is the ability to browse through a graphical representation of an interaction network.

Between the author's work and other work on design of monitors to measure interactive performance there are major differences:

(1) The author takes the viewpoint that the interaction should be the unit of analysis, rather than the program. An example of application for this approach is understanding tasks where a graphical user interface, such as X windows, is used. In such situations there is a very poor correspondence between the boundaries of interactions and the boundaries of programs. Another illustration of the advantages of basing analysis on the interaction is that the activities of each server can be related to the various tasks that made use of the server. This is particularly important in distributed systems, where there are many distributed services offered to clients by servers.

(2) The author's work is intended to allow for the evaluation of sets of interactions as well as for the evaluation of individual interactions. In nearly all other work done in this area, the emphasis has been on the evaluation of individual programs. The ability to evaluate sets of interactions means that analysis of interaction networks can be used for tuning of entire systems.

(3) Techniques have been developed that allow a graphical representation of an interaction network to be displayed, and browsed. Other systems in which computation is represented by a directed graph appear not to provide display and browsing facilities.

The author considered that a practical demonstration was required to show that interaction networks can be recorded and analysed. To this end, a substantial prototype, INMON, was constructed by making extensive modifications to SunOS. The two main components of INMON are: (1) the probes and the event recorder; and (2) the analysis programs. The probe locations and parameters were carefully selected, so as to enable individual tasks to be identified, and to provide a great deal of information on the execution and communication that occurred within a task. All 53 probes and most of the event recorder code are located within the SunOS kernel.

Several tools have been developed to perform analysis functions on task information detected by the probes and recorded by the event recorder. The most significant tools are:

(1) a program that provides state decompositions and summaries of disc accesses and remote procedure calls at task, node, and process levels of aggregation; (2) a program that displays the interaction network of a task; and (3) a program for displaying and browsing through interaction networks, which provides most of the features of the first two programs, plus additional features, such as filtering, and access to detailed information on each event record.

In order to analyse interaction networks, a global timebase of high resolution must be available. Such a timebase may be achieved either by having clocks synchronised while event recording is taking place, or by correcting timestamps during analysis. For INMON, the synchronisation approach was chosen. Because no suitable software was available, a new clock synchronisation program was developed. This program has achieved very good synchronisation using a probabilistic algorithm, and introduces two techniques for use in probabilistic clock synchronisation algorithms that seem to be unusual. First, estimated drift rates were used when corrections of sufficient accuracy could not be determined by message passing and, second, the minimum round trip time was computed in a dynamic fashion.

A number of experiments that showed some of the capabilities of INMON have been described. These experiments showed that INMON can provide valuable information on individual interactions and sets of interactions, and on interactions in a graphical user interface environment.

Although the INMON implementation is substantial, the concept of the interaction network could be implemented on a much larger scale. For this to be possible operating system designers would have to include support for interaction network recording in operating system designs.

In conclusion, taking the interaction as the unit for analysis leads to powerful methods for measurement of interactive performance. These methods are particularly valuable for modern graphical interfaces, through which a user may have several interactions in progress concurrently. The author has developed the concept of the interaction network as a means of representing the processing component of an interaction, and has successfully constructed a monitor that records and analyses interaction networks.

The author's main objective in developing these ideas was to provide tools for performance measurement. However, interaction networks provide such a good description of the execution of a task that they should prove valuable for any situation in which detailed information is required on task execution. Interaction networks could therefore be used in program debugging, and as a tool to help students learn about system behaviour.

Appendix A

Clock Synchronisation Experiments

Several experiments were conducted to check the accuracy achieved in practice by version 2.1 of `uoctimed`. An overview of the environment in which the experiments were performed and of the statistics reported is given in Section A.1. The experiments performed and their results are then described in three groups: those carried out under "typical" workload conditions (Section A.2), those carried out under conditions of heavy workload on the Suns acting as the master and the slave (Section A.3), and those carried out under conditions of high network traffic (Section A.4).

A.1 Environment

The environment in which the experiments were conducted was the Sun network of the Department of Computer Science. When the experiments were conducted, the network was an ethernet network, bridged off the campus backbone. The network consisted of 4 thinwire segments, all connected to the same repeater.

Danzig-Melvin clocks were installed in two Sun 3/50s: tui and weka. The D-M clock in weka was measured as gaining 13μs/s relative to the D-M clock in tui. In all experiments weka acted as the master, and tui as the slave running `uoctimed`.

On each correction `uoctimed` reported the correction that it was making, whether the correction resulted from message passing or drift calculation, and the message rtt (if relevant). While we would like to be able to measure actual synchronisation errors, to do this would require additional hardware support. For this reason, the values reported in this appendix are the corrections made by `uoctimed`. As outlined below, it is reasonable to evaluate the accuracy of a clock synchronisation algorithm based on the corrections it makes. The mean correction should be the same as the drift over the resynchronisation interval, with large variations from the mean indicating periods of poor synchronisation.

In some pathological cases, the correction statistics may be a poor indicator of clock synchronisation accuracy. Consider the case where all but two corrections recorded on tui are around 65μs (the expected drift during the resynchronisation interval), but one is very large (say 250μs), and another very small (say -100μs). These corrections would seem to indicate good synchronisation, with the exception of two resynchronisation intervals. A pathological case is where the clocks were originally quite closely synchronised, the high

correction occurred early in the correction measurements, and the low correction late in the correction measurements. In such a case the large correction would cause tui's clock to get about 200μs ahead of weka's, then remain ahead for a considerable number of average corrections, with the small correction finally returning the clocks to reasonably close synchronisation. No such pathological cases were observed in practice, where in all cases a large correction was immediately followed by an offsetting correction.

At the time when a correction is calculated, the clock of the slave is in error by up to D/2 ± E. The error in reading the master's clock is ± E. Finally D/2 is added to the correction. The range of corrections consistent with a maximum error of E + abs(D/2) is, therefore,
D ± 2E. In the experimental environment used, the correction range -145 to 275μs is consistent with a maximum clock synchronisation error of ±137.5μs.

In addition to the information recorded by `uoctimed`, statistics were recorded every 60 seconds on the average ethernet utilisation over the preceding 60 seconds. Utilisations were calculated by dividing the total number of bytes received by the number of bytes that could have been received in the period had they arrived at ethernet's transmission rate of 10 Mbits/s.

A.2 Typical Workload

Five experiments were performed to assess the accuracy of `uoctimed` under conditions of "typical" workload. Each experiment lasted for 4 hours, and was performed on a week day during normal office hours. Various option settings were tried, as outlined in Table A-1. The table shows also the number of corrections computed from the estimated drift rate, and the number computed from TSTAMP_REPLY messages. The final column contains counts for the number of times message passing was attempted, but for which the minimum rtt achieved was greater than the threshold rtt.

Experiment	Options	Drift corrns	Message pass corrns	Long min rtt
1	-m 1	2	2877	2
2	-m 4	2161	721	0
3	-m 12	2646	236	60
4	-b -m 1	0	2881	10
5	-b -m 4	2161	721	3

Table A-1: Typical workload experiments: options and corrections

Login records show that the workload on tui and weka was light during the monitored periods, and the per/minute ethernet utilisations recorded during the experiments were low. Only Experiment 3 (19 out of 236) and Experiment 5 (12 out of 721) contained any

corrections based on message passing that were computed during periods of ethernet utilisation in excess of 10%.

Over the five experiments, 6970 corrections were calculated from the drift rate, with 6960 in the range 64μs to 67μs, and 10 in the range 68μs to 81μs. Drift rate was recalculated 18 times, with results ranging from 12.95μs/s to 13.56μs/s. The drift rate recalculation and corrections computed from the drift rate worked as expected. The small range of values for corrections calculated from the drift rate indicates that nearly all resynchronisation intervals were very close to five seconds.

The remaining corrections were calculated from the timestamps received in TSTAMP_REPLY messages. First, the situation is considered in which a reply is not received within the threshold rtt. In Experiments 1, 2, and 3 such situations result in a correction calculated from the drift rate being made (the -b flag is not specified). In all cases where corrections were calculated from the drift rate because of a long rtt, the succeeding corrections calculated by message passing were close to 65μs, indicating that little error is introduced by using the drift rate to calculate corrections in such cases.

The 60 samples with long rtt in Experiment 3 are of particular interest, as they occurred on consecutive correction attempts, with the minimum rtt observed ranging from 3.23ms to 3.45ms. Because of the 60 long rtt in succession, and because of the 11 previous drift rate based corrections (as -m 12 was specified) there were 71 corrections between two message based corrections. The subsequent message based corrections were 50, 100, 80, and 30μs, indicating that the clocks remained well synchronised throughout this 6 minute period even though all corrections in this period were based on drift rate.

In Experiment 4, there were 10 corrections calculated from messages with long rtt, and in Experiment 5 there were 3. The distributions of the corrections based on message passing are summarised in Figure A-1. Each column is labelled by the lower bound of the times reported in the column.

In most experiments a very small number of corrections were recorded in the -150, -100, 200 and 250 ranges (note that percentages ≤ 0.3 do not appear in Figure A-1). Of the 7403 corrections, only 3 were outside the range of -145 to 275μs. All 3 were recorded in Experiment 4 (an experiment where the -b option was used), and all 3 resulted from calculating corrections based on TSTAMP_REPLY messages with rtt in excess of the 3.1ms threshold rtt. A 500μs correction was based on a reply with an rtt of 4.57ms. The next correction calculated was -350μs, to compensate for the 500μs correction (both corrections are outside the range of corrections shown in Figure A-1). A 290μs correction was based on a reply with an rtt of 3.92ms.

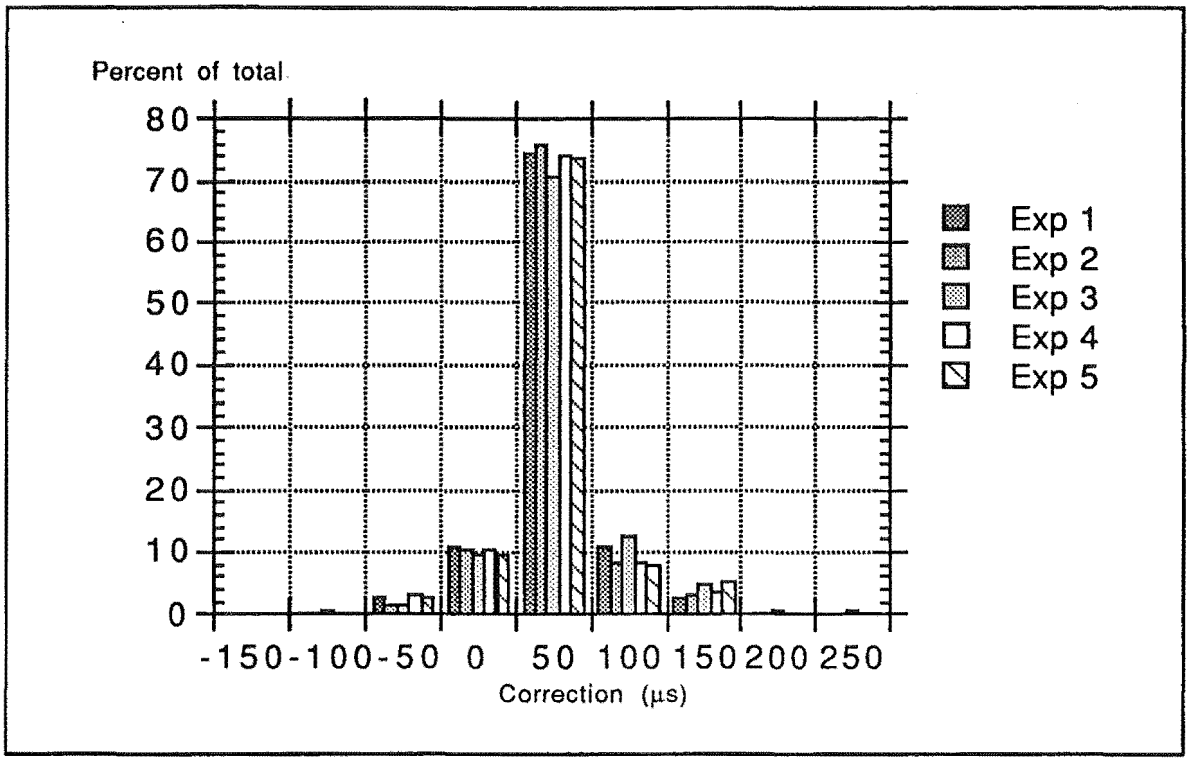


Figure A-1: Distribution of correction values for Experiments 1 to 5

The distributions of corrections were very similar in all 5 experiments. A total of 93.5% of the corrections lie between 0 and 150μs, so tight synchronisation was achieved.

A.3 High Machine Workload

Because tui and weka were under very light workloads during the experiments, it was decided to place an artificial workload on each machine during two further synchronisation experiments by running MUSBUS, a benchmarking suite for Unix systems [MCDO87]. One component of the MUSBUS suite is a multi-user benchmark, where each "user" is a shell script executing a workload of a variety of standard Unix commands.

In this group of experiments, the multi-user MUSBUS benchmark was executed on both tui and weka. The same MUSBUS parameters were used on both machines, with the only changes made from the defaults being to set `nusers` (number of users) to 8, (typing) `rate` to 20, and `iters` (iterations) to 10. Used in this way, MUSBUS created a heavy workload on each machine. The load average for each was observed to be between 9 and 12 for the duration of these experiments, where the load average is the average number of process on the ready list or waiting for I/O to complete, as sampled over the previous

minute [LEFF88]. Also, response was observed to be very sluggish while MUSBUS was operating.

Each of the experiments was performed for one hour. Experiments 6 and 7 were done with options -m 1 and -b -m 1 respectively, with results shown in Figure A-2.

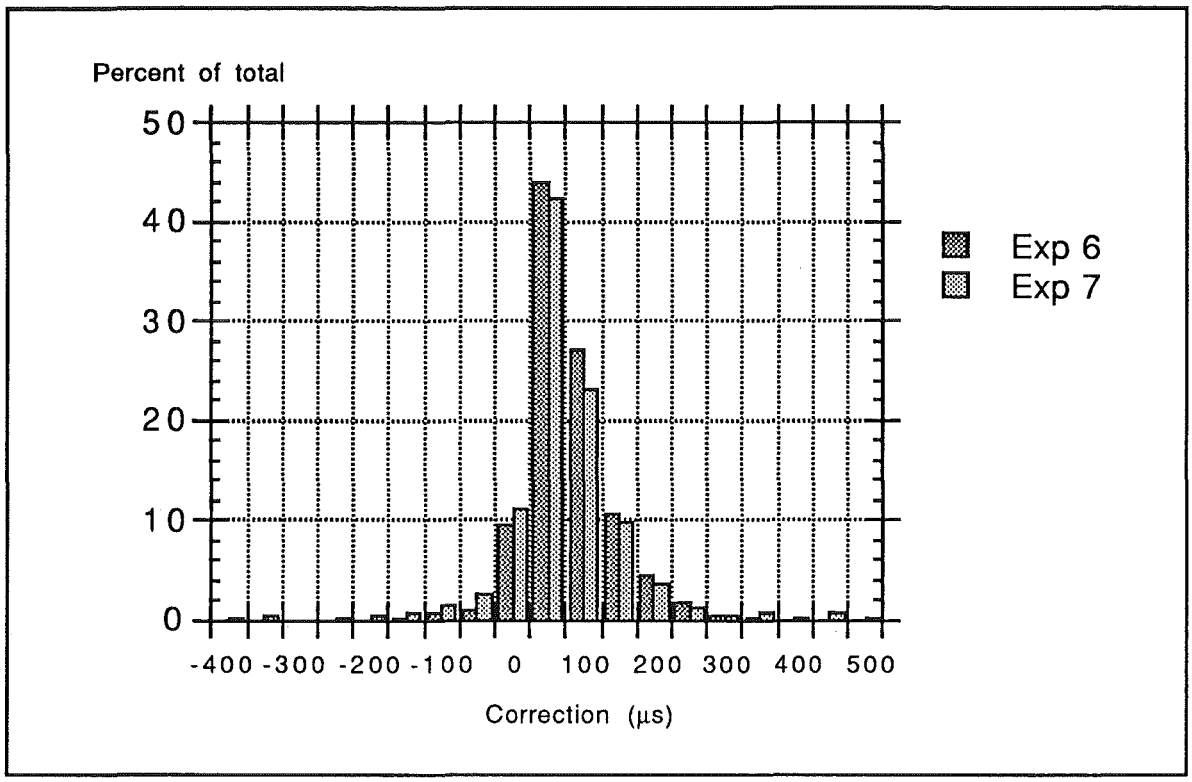


Figure A-2: Correction percentages for Experiments 6-7

Experiments 6 and 7 show that, despite the heavy load, nearly all corrections are in the range expected. Experiment 7 highlights the danger of using the -b option, with a few very large corrections (and corresponding negative adjustments) observed. The six largest corrections recorded in Experiment 7, not shown in Figure A-2, are 4710μs followed by -4400μs (12.56ms rtt), 1100μs followed by -900μs (5.29ms rtt), and 800μs followed by -690μs (4.75ms rtt).

In Experiment 6, 110 of the 517 corrections were based on drift rate. Of these 110, in 4 rtt was between 5 and 6ms (the highest 5.81), in 14 rtt was between 4 and 5ms, and in 82 rtt was between 3.1 and 4ms. In Experiment 7, all corrections were based on message passing (the -b option was used), with 146 of the 543 having rtt above the threshold rtt. Of these 146, one rtt was 12.56ms, one was 5.29ms, 14 were between 4 and 5ms, and 110 were between 3.1 and 4ms.

The distributions of corrections made in Experiments 6 and 7 are skewed toward values above 100μs in comparison with the distributions of Experiments 1-5. The main reason for this is that because of the heavy workload the resynchronisation interval was extended. In Experiment 6, 517 corrections were made in one hour giving an average resynchronisation interval of close to 7s. Therefore many intervals are longer than the 5s target, resulting in more corrections of more than 100μs than would otherwise be expected.

In Experiment 6, five corrections (280, 290, 320, 340, and 390μs) are outside the calculated correction bounds of -145 to 275μs. The bounds were based on a resynchronisation interval of 5s, and given that many of the resynchronisation intervals were considerably longer than 5s, it seems likely that these corrections would have been within bounds calculated for their actual resynchronisation intervals. As resynchronisation interval length was not measured this was not possible to verify.

A.4 High Network Traffic

Network utilisation was relatively low in Experiments 1 to 7, where only 31 of the 7436 message passing corrections were calculated during a period of ethernet utilisation of more than 10%. A number of small experiments were performed to see whether increasing the ethernet load had any affect on the accuracy of `uoctimed`.

The Sun `spray` utility was used to generate ICMP echo and echo reply packets. Spray generates packets at regular intervals, so the type of workload it generates is quite different from the bursty nature of most network workloads. It was felt, however, that if `uoctimed` becomes much less accurate under high network workloads then there should be some indication of this under the type of workload provided by `spray`.

The results of three experiments are reported. All three were run for 10 minutes, and all three used `uoctimed` with the "-m 1" options. In Experiment 8, the machines involved in generating the network load were not tui and weka. In Experiment 9, tui and weka generated the network workload. Experiment 10 was the most severe test, in which weka and tui both executed MUSBUS in the way described in the previous section, and weka and tui both generated network traffic using `spray`.

Time (ms)	Exp 8	Exp 9	Exp 10
3.1 -> 3.99	9	6	33
4 -> 4.99			8
5 -> 5.99			9
6 -> 6.99			12
7 -> 7.99			2
Total	9	6	64

Table A-2: Rtt of replies with rtt above the threshold

In all experiments the ethernet utilisation was substantial: 27% for most of Experiment 8, 16% for all of Experiment 9, and 21% for Experiment 10. The number of responses with long rtt is shown in Table A-2, and a summary of the corrections calculated from messages appears in Figure A-3.

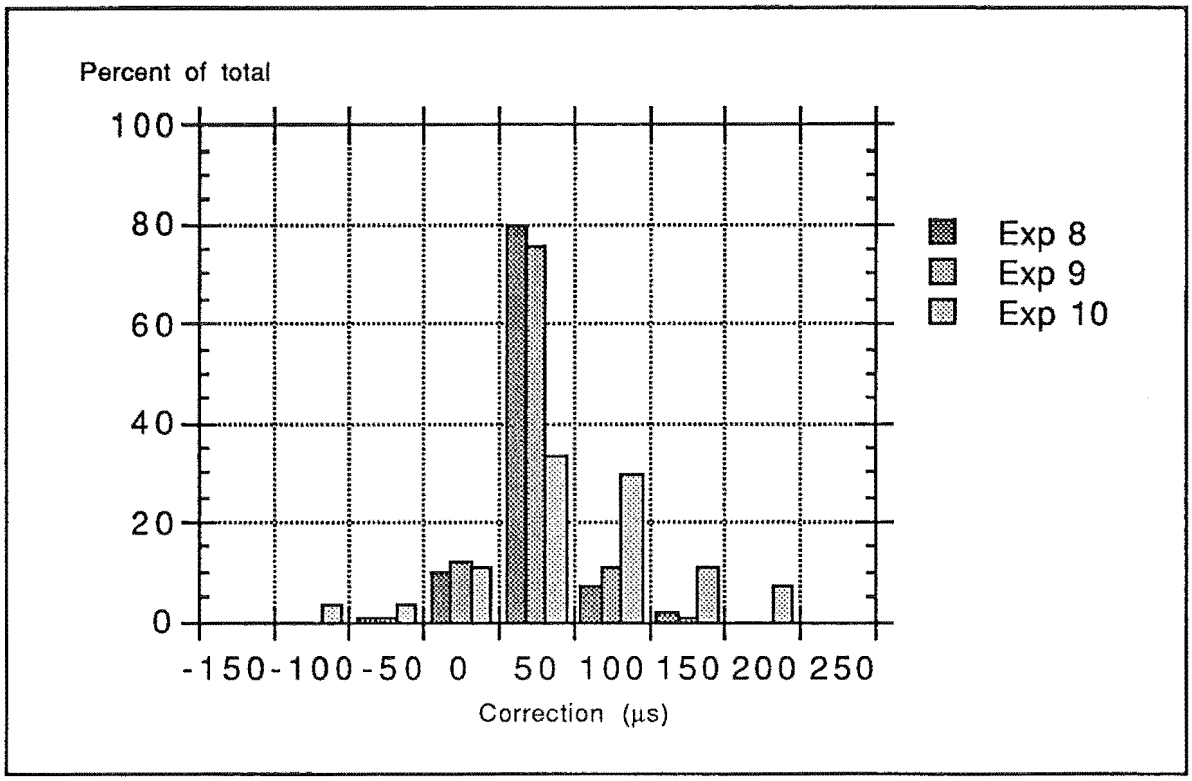


Figure A-3: Correction percentages for Experiments 8-10

The distributions of the corrections made in Experiments 8 and 9 are very similar to those observed in Experiments 1 to 5 under conditions of light network load. The number of replies with long rtt was significantly more than for Experiments 1 to 5, although still less than 10%. The corrections made in Experiment 10 indicate that `uoctimed` performed well under conditions of very heavy workload: heavy workload on each node, heavy ethernet workload, and heavy workload on the components of the kernel that manage the ethernet interface and that perform processing for the IP protocols. The correction distribution indicates good synchronisation, with the distribution skewed in a similar way to the Experiment 6 and 7 distributions. In Experiment 10, 64 of the 91 attempts to synchronise resulted in replies with long rtt, but `uoctimed` still performed well.

Appendix B

Unix Manual Pages

This appendix contains Unix manual pages that describe INMON and the Danzig-Melvin clocks. A brief overview of the manual pages is given in Section B.1. The remaining sections contain the manual pages, with each section of the appendix containing manual pages from a single section of the Unix manual, as summarised in Table B-1.

Appendix section	Unix manual section
B.2	1 (User commands)
B.3	2 (System calls)
B.4	3 (Library functions)
B.5	5 (File formats)
B.6	8 (System programs)

Table B-1: Summary of the sections of Appendix B

B.1 Overview

The manual pages included in this appendix are summarised in Table B-2. Logically, there are three groups of manual pages: those related to Danzig-Melvin clocks (previewed in Subsection B.1.1); those related to the event recording functions of INMON (previewed in Subsection B.1.2); and those related to the analysis functions of INMON (previewed in Subsection B.1.3).

The manual pages included in this appendix are listed in Table B-2 in the order that they appear. The Unix format `entry_name(section_number)` is used for naming manual entries. The manual page `tmrLib(3)` describe the functions: `tmrOpen()`, `tmrPrescaleBy()`, `tmrMap()`, `tmrTimeval()`, `getTS()`, and `getTV()`. Finally, note that manual entries with the name `perf` occur in Unix manual sections 2 and 8.

B.1.1 Time-related Manual Pages

`uoctimed(1)` describes the clock synchronisation program used in INMON. System calls that read and change a Danzig-Melvin (D-M) clock are covered in `adjtmr(2)`, `gettmr(2)`, and `settmr(2)`. Library functions that are related to use of a D-M clock are described in `tmrLib(3)`. A program that initialises a D-M clock is covered in `utimeset(1)`.

Appendix subsection	Unix manual page
B.2.1	analyse(1)
B.2.2	insplit(1)
B.2.3	perfdump(1)
B.2.4	pfilter(1)
B.2.5	stripserver(1)
B.2.6	toxgrab(1)
B.2.7	uoctimed(1)
B.2.8	utimeset(1)
B.3.1	adjtmr(2)
B.3.2	getnextstn(2)
B.3.3	getperfstat(2)
B.3.4	gettmr(2)
B.3.5	perf(2)
B.3.6	perfon(2)
B.3.7	perfsvc(2)
B.3.8	settmr(2)
B.4.1	tmrLib(3)
B.5.1	perf_data(5)
B.5.2	perf.log(5)
B.6.1	dumpperfstat(8)
B.6.2	perf(8)
B.6.3	perf_warn(8)
B.6.4	perfd(8)

Table B-2: Summary of the subsections of Appendix B

B.1.2 Manual Pages for the INMON Event Recorder

`perfd(8)` describes the program that performs the event recording. System calls used by `perfd` are discussed in `perfon(2)` and `perfsvc(2)`. The format of log files produced by `perfd` is described in `perf.log(5)`. `perf_warn(8)` describes a program used by `perfd` to report exceptional conditions. `perf(8)` covers a program that can be used to control `perfd`. Information required by `perf` is stored in a file whose format is described in `perf_data(5)`. User programs can have events recorded by `perfd` using the system call described in `perf(2)`. Counts of unusual conditions that can occur within INMON are reported by the program described in `dumpperfstat(8)`, which uses the system call described in `getperfstat(2)` to get the counts from the kernel.

The system call described in `getnextstn(2)` returns a freshly generated sub-task number.

B.1.3 Manual Pages for the INMON Analysis Programs

The INMON analysis programs are described in: `analyse(1)`, `insplit(1)`, `perfdump(1)`, `pfilter(1)`, `stripserver(1)`, and `toxgrab(1)`. `Xgrab` is described in [BARN89], and `inbrowser` in [THOO91].

B.2 User Commands

B.2.1 analyse

NAME

analyse - summarise performance information for the specified task log files

SYNOPSIS

analyse [**-a**] [**-s**] [**-e**] [**-r**] [**-d**] [**-v**] [**-f**] *logfile* [*logfile* ...]

DESCRIPTION

analyse summarises performance information for the interaction networks in *logfiles*, with the summary written to stdout. Each task log file must be one produced by **insplit(1)**, and therefore will contain one interaction network. The activities recorded in an interaction network are of two types: process-related and message-related. Information can be summarised for all events and activities in an interaction network, and/or for the events and activities on the critical path only. Also summaries can be prepared at three different levels:

Process — Process-related information is summarised for each process, and message-related information is summarised for each pair of communicating processes.

Machine — Process-related information is summarised for each machine, and message-related information is summarised for each pair of communicating machines.

Overall — All process-related information is summarised in a single report, and all message-related information is summarised in a single report.

This means that there 12 possible reports due to the combinations of type of information (process, message), events and activities selected (all, critical path), and summary level (process, machine, overall). **analyse** asks the user which of these 12 reports it is to produce in the following way. If the **-a** option is specified then all 12 reports will be produced. Otherwise the user is prompted 3 times: once to ask which Overall reports are to be produced, once to ask which Machine reports are to be produced, and once to ask which Process reports are to be produced. Prompts are written to stderr, and input is read from stdin. In response to each prompt the user specifies which reports (if any) at the specified level are to be produced. Valid response lines consist of the following (case sensitive) letter combinations, separated by white space, followed by newline:

a — Produce all 4 reports.

p — Produce both process reports.

m — Produce both message reports.

i — Produce both entire interaction network reports.

c — Produce both critical path reports.

ip Produce the entire interaction network process report.

im Produce the entire interaction network message report.

cp Produce the critical path process report.

cm Produce the critical path message report.

The reports produced for each level are the inclusive or of those specified. If none are specified, then none are produced for that level.

The remaining options control the information sections present in the reports. By default no sections are included. Note that all times printed in reports are in units of seconds.

OPTIONS

-e Event counts (process, message). This report section summarises the number of events recorded for the processes or messages being summarised. The format for this report is the same for both the process report and the message report. For each type of event that occurred at least once in the processes or messages, the event count and the event name are printed. Events are listed in event number order. Each event in an interaction network is either counted as a process event or a message event. Process-related events are counted for the process in which they occur. Message-related events are counted for the <source, destination> process pair of the message the event is associated with.

NOTE: if a fork occurs in a message and the sub-messages are delivered to two or more processes, then all of the pre-fork events and activities have the destination process of the first part of the message as their destination process. This allocation is probably not ideal, but this situation is very unlikely, and **analyse** prints a warning message if it detects it.

ALSO NOTE: it could be argued that events which involve a process and a message should be included in both a process count and a message count.

-s State counts, State times (process, message). Each activity in an interaction network represents a period during which the associated process or message is in a particular state. This report summarises the number of times the processes or messages being summarised were in a particular state, and the total amount of time spent in each state. For messages the states are:

queued Message is queued within a machine.

network Message is in transit across a network.

For processes the states are:

run Process is executing on the CPU.

ready Process is on CPU ready list.

created Process is being created by its parent.

zombie	Process has exited and is waiting for its parent.
swapped	Process is swapped.
sleep_terminput	Process is awaiting terminal input.
sleep_child_wait	Process is waiting for a child to exit or change status.
sleep_sock_send	Process is waiting for space to become available so that it can write to a stream socket.
sleep_sock_rcv	Process is waiting for a message to arrive on a socket.
sleep_RPC_clnt	Process acting as an RPC client is waiting for a reply to its remote call.
sleep_RPC_svc	Process acting as an RPC server is waiting for a remote call.
sleep_accept	Process is waiting for the arrival of an incoming connect on a socket willing to accept connections.
sleep_disc	Process is waiting for the completion of a disc request.
sleep_other	Process is waiting for some reason other than those listed above.

Counts and times are printed only for states that occur in the processes or messages in question.

-d Disc usage (process). A summary of disc access counts and time spent waiting for disc requests is presented. A summary is printed for each drive accessed. This summary contains details for reads and writes to each of the partitions of a drive that were accessed by the processes in question. Totals for access times and access counts are reported for each disc. Overall totals appear in the state counts and times section.

-r RPC client statistics, RPC server statistics (process). If one of the processes in question has acted as an RPC client, then RPC client call count and reply wait time summaries are presented grouped by remote program number, with call counts and elapsed times presented for each procedure called. If one of the processes in question has acted as a server then an RPC server call count summary is presented. At present programs and procedures are identified by their numbers.

-v Process validation (process, Process level only). This report is only produced for processes that are recorded in their entirety in an interaction network. The report compares various pieces of information recorded in a process' rusage structure (see **wait3(2)**) at the time that it exited, against the same pieces of information calculated by analysing information in the interaction network.

The number of voluntary context switches, involuntary context switches, message receives, and swap out operations

should be the same as calculated by both methods. The only exception is if a process is sent a signal which stopped it, in which case the number of involuntary context switches recorded in the rusage structure will be greater than the number calculated from the interaction network by the number of times a process was stopped in this way.

In the absence of errors, the number of message sends calculated by each method should be the same. The message send counter is incremented on every attempt to send a message, whereas the interaction network only records successful message sends. Therefore in the presence of errors the number of message sends recorded in the rusage structure will be greater than the number calculated from the interaction network.

The CPU times are recorded in quite different ways. The time in the rusage structure is calculated by sampling. On every clock interrupt (every 20ms on Sun 3s) the CPU time of the running process is updated. For interaction networks, CPU time is calculated from event records. The start and stop times of each burst of CPU usage are recorded, and the lengths of the CPU bursts are totalled to get the CPU time. The CPU time calculated from the interaction network should be much more accurate, particularly if microsecond timers are installed.

-f Produce all of the above reports.

The reports produced are the inclusive or of the reports specified. **analyse** is quite happy for options to be specified more than once, and for other report options to be present as well as the **-f** option.

SEE ALSO

insplit(1), wait3(2), perfd(8)

BUGS

The activity and event selection mechanisms could be improved.

Could show overall totals across all discs in a disc summary, broken down into read and write components.

Could try to include validation of disc and NFS information recorded in interaction networks against the inblock, oublock, and majflt fields of an rusage structure.

B.2.2 insplit

NAME

insplit - groups the event records in the specified node log file(s) into task log file(s)

SYNOPSIS

insplit [**-c** *cutoff*] [**-o** *outputdir*] [**-u** *undeffile*] *logfile* [*logfile* ...]

DESCRIPTION

insplit reads the node log files specified, and for each interaction network identified from these log files produces a task log file. The name given to a task log file is the sub-task number of the initial sub-task of the task. See **perf.log(5)** for more information on node and task log files.

insplit makes some changes to the events read in from the node log file(s). For each _INPUT_END event record a PERF_SINK event record is added to record the end of the sub-task previously associated with the process receiving the input. Each PERF_SOCK_SBREL event record is replaced by an appropriate number of PERF_SINK event records. Each PERF_SOCK_STREAM_RCV event record is replaced by
N-1 PERF_SINK event records for the first N-1 sub-task numbers in the PERF_SOCK_STREAM_RCV event record, and a PERF_PE_DEQUEUE event for the Nth sub-task number.

OPTIONS

-c *cutoff*
: Only interaction networks consisting of more than *cutoff* (default 2) event records are output.

-o *outputdir*
: Task log files are written into *outputdir* (default .).

-u *undeffile*
: All event records that are not allocated to any interaction network, are recorded in a log file called *undeffile* (default is not to produce this file).

SEE ALSO

perf.log(5)

B.2.3 perfdump

NAME

perfdump - produce a text description of the event records in a node log file or a task log file

SYNOPSIS

perfdump [**-r**] *logfile*

DESCRIPTION

perfdump prints a description of each event record in the node or task log file *logfile*. For task log files the internet number of each event record is also printed.

The **-r** option specifies how **perfdump** should handle PERF_SOCK_STREAM_RCV events that are recorded over more than one event record. By default, a single aggregated event record is described in these situations. If the **-r** option is specified the individual event records are described.

SEE ALSO

perf.log(5)

B.2.4 pfilter

NAME

pfilter - removes event records of the specified type(s) from a node or task log file

SYNOPSIS

pfilter *logfile event ...*

DESCRIPTION

pfilter copies all event records, except those of the specified type(s) from *logfile* to *logfile*.filter. For task log files, the file *logfile*.filter.inetnum contains internet numbers for the event records in *logfile*.filter. Events are identified by numbers (see <sys/perform.h>).

EXAMPLES

To remove the three CPU-related events (PERF_SETRQ, PERF_SWTCH, and PERF_RESUME) from the log file 040000a9:

```
example% pfilter 040000a9 27 28 29
```

SEE ALSO

perf.log(5)

WARNING

Removing events of any type other than simple will destroy the structure of interaction network(s) present in *logfile*. The intended use of **pfilter** is to remove frequently occurring simple events, particularly the CPU-related events, from a log file.

B.2.5 stripserver

NAME

stripserver - removes from a task log file all event records in a sub-task that follow a PERF_RPCSVC_WAIT event record

SYNOPSIS

stripserver *logfile* [*logfile* ...]

DESCRIPTION

For each task log file *logfile* specified, **stripserver** copies all event records, except those in a sub-task following a PERF_RPCSVC_WAIT event record, from *logfile* to *logfile*.ss. Internet numbers are recorded in *logfile*.ss.inetnum.

The reason for the existence of **stripserver** is that when a server is used (often nfsd) then the interaction's association with the server process is over once the server has finished the processing for a particular request, that is at a PERF_RPCSVC_WAIT event. In the current INMON implementation, subsequent event records are associated with the server process until a new request arrives, resulting in many spurious events being recorded as part of an interaction after the interaction has finished using the server process. This problem is accentuated by the fact that every nfsd process is awoken when a request arrives, causing most of these processes to be scheduled and then to go back to sleep. **stripserver** removes these unwanted events.

INMON could be changed so that when a PERF_RPCSVC_WAIT event occurs, the sub-task number of the current process is reset to invalid.

SEE ALSO

perf.log(5)

B.2.6 toxgrab

NAME

toxgrab - produces from a task log file a .xgrab file to allow display of an interaction network

SYNOPSIS

toxgrab [**-e** *event_space*] [**-o** *outputdir*] [**-p** *proc_space*] [**-s** *scaler*] *logfile* ...

DESCRIPTION

For each task log file specified, **toxgrab** produces a file, *logfile*.xgrab, which can be displayed using **xgrab(1)**. In an .xgrab file, each event record in the *logfile* from which the .xgrab file was produced is represented by a vertex. The (X,Y) coordinates of the vertex are calculated as described below, and the name of the vertex is the event's sequence number, as reported by **perfdump(1)**. Also present in the .xgrab file are definitions of the edges of the interaction network, and of a label vertex for each process involved in the interaction. Each label vertex has as its name a 6 digit hex number, with the first two digits being the low order byte of the internet number of the machine on which the process executed, and the remaining 4 digits being the pid of the process.

The coordinates of a vertex are calculated as follows. The X-coordinate is based on whether a vertex is process-related or message-related. For process-related events the vertex is given the X-coordinate of its process. The default process spacing is 15. For message-related vertices the X-coordinate is between the X-coordinates of the process end points of the message, and is calculated to be the same proportion between the two end points as the Y-coordinate is between the Y-coordinates of the process end point events.

The Y-coordinate is calculated based either on time or on an equal spacing of events that preserves the partial order. In the time-based method, the Y-coordinate of each vertex is calculated as the negative of the number of milliseconds since the source vertex of the network, divided by *scaler* (whose default is 60). In the partial order method, the Y-coordinate of each vertex is *event_space* (default 15) less than the minimum value of the Y-coordinates of the predecessors of the vertex. This gives a reasonably uniform spacing with the partial ordering of events in the network preserved.

OPTIONS

-e *event_space*
: Vertical spacing between events in the equal spacing mode (default 15).

-o *outputdir*
: Xgrab files are written into *outputdir* (default .).

-p *proc_space*
: Horizontal spacing between events in adjacent processes (default 15).

-s *scaler*
Scaler used in algorithm for calculating Y-coordinates in time based mode (default 60). If *scaler* is 0 then equal spacing mode is used (time based is the default).

SEE ALSO
perfdump(1), xgrab(1), perf.log(5)

B.2.7 uoctimed

NAME

uoctimed - clock synchronisation daemon

SYNOPSIS

uoctimed [**-b**] [**-d**] [**-m** *message_freq*] [**-n**] [**-r** *resynch_period*] *host max_error*

DESCRIPTION

uoctimed is a clock synchronisation daemon. **uoctimed** synchronises the time of the machine it is running on to that of *host*. Note that a set of kernel modifications to the ICMP code should be installed in both the local machine and *host*.

max_error specifies the maximum error, in microseconds, which **uoctimed** will allow to develop between the clock of the machine on which **uoctimed** is running and that of *host*. Usually, **uoctimed** will do considerably better than this maximum for most of the time.

The **-r** flag can be used to specify the interval between corrections (default 5 seconds).

The **-m** flag can be used to specify the frequency with which **uoctimed** uses message passing to determine a correction. If *message_freq* is 1 (the default) message passing is attempted on every correction. If it is 2, then message passing is attempted on every second correction, if 3 on every third correction, and so on. Corrections not based on message passing are based on the estimated drift rate between the two clocks, which is recalculated every hour. If no drift file exists, then for the first hour the drift is assumed to be 0. The drift rate is then recorded in a drift file (/etc/uoctimed.drift.*host*) to be available for subsequent executions of **uoctimed**.

The default where a message passing correction is attempted, but the resulting value is deemed unreliable for the target maximum error, is to base that particular correction on the estimated drift rate. If the **-b** flag is specified then the value calculated from message passing is used in these cases despite the sometimes large errors in corrections that can result.

If the **-n** flag is specified then uoctimed does not fork off a child to run as the daemon. If the **-d** flag is specified then debugging output is written to stderr. Both of these options are primarily debugging aids.

uoctimed synchronises Danzig-Melvin (D-M) microsecond timers. For **uoctimed** to operate correctly, the D-M timers on both local and remote hosts must be in "timeval" mode. **utimeset(1)** has been written to put a D-M timer into "timeval" mode, and to set it to the current system time.

FILES

/etc/uoctimed.drift.*host*

SEE ALSO

utimeset(1)

B.2.8 utimeset

NAME

utimeset - put the microsecond timer into "timeval" mode, and set it to the current system time

SYNOPSIS

utimeset

DESCRIPTION

utimeset puts the microsecond timer into "timeval" mode, and sets it to the current system time.

SEE ALSO

gettmr(2), settmr(2), tmrLib(3)

B.3 System Calls

B.3.1 adjtmr

NAME

adjtmr - adjust the Danzig-Melvin microsecond timer

SYNOPSIS

```
#include  <gettmr.h>

int  adjtmr(tv)
struct  timeval  *tv;
```

DESCRIPTION

adjtmr() adjusts the microsecond timer by the amount of time specified by *tv*. Note that the signs of *tv*->tv_sec and *tv*->tv_usec should be the same, otherwise the results of the call will be unpredictable.

If INMON event recording is in progress (see **perfd(1)**) the magnitude of the change is restricted so as not to perturb too greatly timestamps recorded by INMON. Currently, changes when INMON event recording is enabled are capped at +/- 1 milliseconds, with changes greater than that silently reduced in magnitude.

adjtmr() does not change the value of the timer directly, it simply changes an offset variable. The value of this offset is used by **gettmr(2)** to adjust "timeval" mode readings. The offset is set to 0 by **settmr(2)**.

For **adjtmr()** to function correctly the microsecond timer must be in "timeval" mode (see **gettmr(2)**).

RETURN VALUES

adjtmr() returns 0 on success. On failure, it returns -1 and sets **errno** to indicate the error.

ERRORS

EINVAL	*tv* is an invalid address
EPERM	Neither the process's effective or real user ID is superuser.

SEE ALSO

gettmr(2), settmr(2), perfd(8)

B.3.2 getnextstn

NAME

getnextstn - get the next available sub-task number

SYNOPSIS

```
#include  <sys/perform.h>

ipid_t  getnextstn()
```

DESCRIPTION

getnextstn() returns the next sub-task number in the sequence. Note this call does not change the sub-task number of the current process.

RETURN VALUES

getnextstn() returns a new sub-task number. This call always succeeds.

B.3.3 getperfstat

NAME

getperfstat - retrieve a copy of the perf_stat structure maintained by INMON within the kernel

SYNOPSIS

```
#include  <sys/types.h>
#include  <sys/time.h>
#include  <sys/perform.h>

int  getperfstat(pp)
struct  perf_stat  *pp;
```

DESCRIPTION

getperfstat() retrieves the perf_stat structure from the kernel, and stores a copy where *pp* points. The contents of the structure are described in **dumpperfstat(8)**.

RETURN VALUES

getperfstat() returns 0 on success. On failure, it returns -1 and sets **errno** to indicate the error.

ERRORS

EINVAL *pp* points outside the process address space.

SEE ALSO

dumpperfstat(8)

B.3.4 gettmr

NAME

gettmr - read the time from the Danzig-Melvin microsecond timer

SYNOPSIS

```
#include <gettmr.h>

int gettmr(cnt, time)
int cnt;
unsigned char *time;
```

DESCRIPTION

gettmr() provides an interface to kernel versions of **getTS(3)** and **getTV(3)**. One advantage of using **gettmr()** is that within the kernel these functions are performed at the splimp() interrupt level, so there is no possibility that the time read will be incorrect due to an interrupt occurring. The system call overhead is quite considerable, however, compared to direct calls to **getTS(3)** and **getTV(3)**.

To have a **getTS(3)** performed, *cnt* should be 4, and *time* should point to a 4 byte area (usually a variable of type unsigned) to hold the result.

To have a **getTV(3)** performed, *cnt* should be 8, and *time* should point to an 8 byte area (usually a variable of type struct timeval) to hold the result. To use the getTV variant the clock should be in "timeval" mode, which can be achieved by calling **tmrTimeval(3)**.

If the clock is to be always used in timeval mode, then it is suggested that **utimeset(1)** be executed in **rc.local(8)**. **utimeset(1)** contains a call to **tmrTimeval(3)**. Thereafter the microsecond timer should be left in "timeval" mode, and **gettmr()**, **settmr(2)**, and **adjtmr(2)** can all be called safe in the knowledge that the microsecond timer is in "timeval" mode.

RETURN VALUES

gettmr() returns 0 on success. On failure, it returns -1 and sets **errno** to indicate the error.

ERRORS

EINVAL *cnt* has a value other than 4 or 8, or *time* is an invalid address.

SEE ALSO

utimeset(1), **adjtmr(2)**, **settmr(2)**, **getTS(3)**, **getTV(3)**, **tmrTimeval(3)**, **rc.local(8)**

B.3.5 perf

NAME

perf - request that an event record be written to the current node log file

SYNOPSIS

```
#include  <sys/types.h>
#include  <sys/time.h>
#include  <sys/perform.h>

int perf (record, set_stn, stn)
struct perf_record *record;
int set_stn;
ipid_t stn;
```

DESCRIPTION

perf() writes an event record to the current node log file. The data pointed to by *record* is written to the log file. The data should be a well-formed event record as described in **perf.log(5)**. The kernel performs a minimal check on the data before writing it to the node log file.

If *set_stn* is non-zero, and the record is recorded successfully, the sub-task number of the calling process is set to *stn*. Otherwise, the value of *stn* is ignored.

If event recording is not in progress then the event record (and any request to set the process sub-task number) are simply ignored, although 0 is still returned.

RETURN VALUES

perf() returns 0 on success. On failure, it returns -1 and sets **errno** to indicate the error.

ERRORS

EFAULT	*record* points outside the process's allocated address space.
EINVAL	The length specified in the event record is too short, or more than **MAXPERFDATA**.
EPERM	The process's effective user ID is not super-user.

SEE ALSO

perfsvc(2), perf.log(5), perfd(8)

B.3.6 perfon

NAME

perfon - turn event recording on or off

SYNOPSIS

```
#include  <sys/types.h>
#include  <sys/time.h>
#include  <sys/perform.h>

int  perfon  (condition,  events,  event_len)
int  condition;
char  *events;
int  event_len;
```

DESCRIPTION

perfon() sets event recording on or off, as specified by *condition*. Legitimate values for *condition* are:

PERF_ON	logging is to be done
PERF_OFF	logging is not to be done

Only the super-user may successfully execute this call.

The values of *events* and *event_len* are used only where the value of *condition* is PERF_ON. *events* is an array of *event_len* characters, specifying which of the events 0 to *event_len*-1 should be recorded. Events with a non-zero value in *events* are recorded, as are all events with numbers greater than or equal to *event_len*.

If *events* is a null pointer, or *event_len* is 0 then recording of all events is enabled.

RETURN VALUES

perfon() returns the previous value of the event logging condition on success. On failure, it returns -1 and sets **errno** to indicate the error.

ERRORS

EINVAL	The value of *condition* is outside the range of valid values, or *events* is an invalid pointer, or *event_len* is negative or greater than MAX_PERFRECTYPE + 1.
EPERM	Neither the process's effective or real user ID is superuser.

SEE ALSO

perf(2), perfd(8)

B.3.7 perfsvc

NAME

perfsvc - write event records to the specified file descriptor

SYNOPSIS

```
int  perfsvc(fd,  limit)
int  fd;
int  limit;
```

DESCRIPTION

perfsvc() specifies the node log file to the kernel. The kernel writes performance event records to this file until an exceptional condition occurs, at which time the call to **perfsvc()** returns. *fd* is a file descriptor that identifies the node log file. Programs should open this file for writing before calling **perfsvc()**. *limit* has a value between 0 and 100, instructing **perfsvc()** to return when the percentage of free disk space on the filesystem that contains the node log file drops below this limit. Thus, the invoking program can take action to avoid running out of disk space. **perfsvc()** returns when one of the following conditions occurs:

- The process receives a signal that is not blocked or ignored.
- An error is encountered while writing to the node log file.
- The minimum free space (as specified by *limit*), has been reached.

Only processes with a real or effective user ID of superuser may execute this call successfully.

RETURN VALUES

perfsvc() returns only on an error, with -1 returned, and **errno** set to indicate the error.

ERRORS

EAGAIN — *fd* referred to a stream, that was marked for System V-style non-blocking I/O, and no data could be written immediately.

EBADF — *fd* is not a valid descriptor open for writing.

EBUSY — Another process is currently executing within **perfsvc()**.

EDQUOT — The user's quota of disk blocks on the file system containing the node log file has been exhausted.

Free space on the file system of the node log file has fallen to less than *limit* percent.

EFBIG — An attempt was made to write a file that exceeds the process's file size limit or the maximum file size.

EINTR — The call is forced to terminate prematurely due to the arrival of a signal whose **SV_INTERRUPT** bit in **sv_flags** is set (see **sigvec(2)**). **signal(3V)**, in the System V compatibility library, sets this bit for any signal it catches.

EINVAL Event logging is disabled (see **perfon(2)**).

fd does not refer to a file of an appropriate type. Regular files are always appropriate.

EIO An I/O error occurred while reading from or writing to the node log file.

ENOSPC There is no free space remaining on the file system containing the log file.

ENXIO A hangup occurred on the stream being written to.

EPERM The process's effective or real user ID is not superuser.

EWOULDBLOCK The file was marked for 4.2BSD-style non-blocking I/O, and no data could be written immediately.

SEE ALSO

perf(2), perfon(2), sigvec(2), signal(3V), perf.log(5), perfd(8)

B.3.8 settmr

NAME

settmr - set the Danzig-Melvin microsecond timer

SYNOPSIS

```
#include  <gettmr.h>

int  settmr(tv)
struct  timeval  *tv;
```

DESCRIPTION

settmr() sets the microsecond timer to the time specified by *tv*.

For **settmr()** to function correctly the microsecond timer must be in "timeval" mode (see **gettmr(2)**).

RETURN VALUES

settmr() returns 0 on success, -1 and **errno** set on failure.

ERRORS

EINVAL	*tv* is an invalid address.
EPERM	The process's effective or real user ID is not superuser.

SEE ALSO

adjtmr(2), gettmr(2), perfd(8)

B.4 Library Functions

B.4.1 tmrLib

NAME

tmrLib - functions to manipulate the Danzig-Melvin microsecond timer

SYNOPSIS

```
#include  <sys/types.h>
#include  <sundev/tmrreg.h>
#include  <gettmr.h>

int  tmrOpen()

tmrPrescaleBy(x)
unsigned  short  x;

tmrMap()

tmrTimeval()

getTS(x)
unsigned  *x;

getTV(x)
struct  timeval  *x;
```

DESCRIPTION

The Danzig-Melvin timer is a 64 bit timer with a maximum clock rate of 4 MHz. The rate at which the clock ticks can be set by **tmrPrescaleBy()**, which sets the tick rate to 4 / *x* MHz. **tmrOpen()** opens the clock device (/dev/tmr0) for reading and writing, initialises the timer, and returns the resulting file descriptor. The 64 bits of the timer can be accessed using **read(2)** and **write(2)** on the descriptor returned (most significant byte of the timer is the first byte of the device file). Note that the functions **tmrPrescaleBy()**, **tmrMap()**, and **tmrTimeval()** can only be called after **tmrOpen()** has been called, and before the descriptor that it returns has been closed.

tmrMap() maps the clock into the calling process' address space. This function must be called before use of **getTS()** and **getTV()**.

tmrTimeval() first calls **tmrPrescaleBy()** with a value for *x* of 4, to set the clock rate to 1 MHz (that is 1 tick per microsecond), and then sets the microsecond timer to the current system time. **tmrTimeval()** should be called before calls to **getTV()**.

getTS() reads the low order 32 bits of the timer into the area pointed to by *x*.

getTV() reads the current time of the timer into a unix timeval structure pointed to by *x*.

RETURN VALUES

tmrOpen() returns the file descriptor resulting from the open of /dev/tmr0 on success, returns -1 on error.

tmrPrescaleBy() prints a message and exits on error.

tmrMap() prints a message and exits on error.

tmrTimeval() prints a message and exits on error.

getTS() and **getTV()** have no return value or error conditions.

FILES

/etc/tmr0

SEE ALSO

close(2), gettmr(2), read(2), write(2), exit(3)

B.5 File Formats

B.5.1 perf_data

NAME

perf_data - information on the current perf daemon

SYNOPSIS

/etc/performance/perf_data

DESCRIPTION

perf_data contains the process ID of the perf daemon, and the pathname of the current node log file. The format of the file is

<pid>:<pathname>

where *pid* is the process ID of the current perf daemon, and *pathname* is the full pathname for the current node log file.

EXAMPLE

274:/etc/performance/logfiles/19910327212637.not_terminated.tui

FILES

/etc/performance/perf_data

SEE ALSO

perf(2), perf(8), perfd(8)

B.5.2 perf.log

NAME

perf.log - format of node and task log files

SYNOPSIS

```
#include  <sys/types.h>
#include  <sys/time.h>
#include  <sys/perform.h>
```

DESCRIPTION

Log files begin with a header record consisting of a **perf_header** structure, followed by the name of the previous log file. The name is of length *ph_namelen*. For the first node log file recorded by **perfd(8)**, and for all task log files, the previous node log file name is the empty character string.

```
struct  perf_header  {
    int     ph_magic;       /*  magic number  */
    time_t  ph_time;        /*  time file created  */
    u_long  ph_inetnum;     /*  internet #  of host  */
    u_long  ph_totdrops;    /*  #  of events dropped  */
    short   ph_namelen;     /*  length  of file name  */
};
```

Following the header, there are one or more event records. Each event record has as a header a **perf_record** structure. Values of all of the event-independent parameters are stored in this structure.

```
struct  perf_record  {
    short   pe_record_size;  /*  size  of record  */
    short   pe_event;        /*  the event type  */
    time_t  pe_time;         /*  time  event  recorded*/
    short   pe_pid;          /*  process id  */
    short   pe_param_count;  /*  #  of parameters  */
    ipid_t  pe_stn;          /*  sub-task  number  */
};
```

Immediately following the header is an array of *pe_param_count* two byte integers. The elements of the array give the lengths of the event-dependent parameters of the event record. The event-dependent parameters follow the array, with the length of the first parameter given by the first element of the array, and so on. The length of each event-dependent parameter is a multiple of 2 bytes.

The last event record in a log file must be a PERF_TRAILER event record.

For task log files, a second file exists that contains for each event record in the task log file the internet number of the node on which the event occurred, stored as a 4 byte integer. The name of this second file is the name of the task log file with ".inetnum" appended to it.

SEE ALSO

perf(1), perfd(8)

B.6 System Programs

B.6.1 dumpperfstat

NAME

dumpperfstat - print the INMON "abnormal event" counters

SYNOPSIS

dumpperfstat

DESCRIPTION

dumpperfstat prints the current contents of the perf_stat structure maintained within the kernel by INMON. Each element in the structure is a counter of the number of times that some type of "abnormal event" has occurred. The structure is defined (in <sys/perform.h>) as:

```
struct perf_stat {
    u_long pe_eventdrops;
    u_long pe_toolongdrops;
    u_long pe_zerodequeues;
    u_long pe_shortqueues;
    u_long pe_zerolendeletembufs;
    u_long pe_iploststn;
    u_long pe_sndlowat;
    u_long pe_rcvlowat;
    u_long pe_sndtimeo;
    u_long pe_rcvtimeo;
    u_long pe_appendrecordcalls;
    u_long pe_enqueueallocfailed;
};
```

pe_eventdrops is the number of event records dropped because of lack of event buffers.

pe_toolongdrops is the number of times an event record was dropped because it didn't fit into a long event buffer. These buffers are currently 1000 bytes long.

pe_zerodequeues is the number of times the kernel function pe_dequeue is called with a null queue pointer.

pe_shortqueues is the number of times that an <stn,length> queue has been prematurely exhausted during a pe_dequeue.

pe_zerolendeletembufs is the number of times that an mbuf with an m_len of 0 is discovered while parsing an mbuf chain in pe_deletequeue.

pe_iploststn is the number of times that ip_insertoptions failed to allocate space for options when a sub-task number was to appear in the options, meaning that the packet was sent without the sub-task number.

pe_sndlowat, *pe_rcvlowat*, *pe_sndtimeo*, and *pe_rcvtimeo* are the number of times that the fields so_snd.sb_lowat, so_rcv.sb_lowat, so_snd.sb_timeo, and so_rcv.sb_timeo respectively would have been modified by sosetopt, which is a kernel routine called as a result of a **setsockopt(2)** call. These 4 fields are currently unused

by the socket implementation, and have been taken over by INMON. The sosetopt function is the only place in the kernel where these fields are set, and incrementing these counters has replaced the actual setting code.

pe_appendrecordcalls is the number of times that the kernel function sbappendrecord is called. The interest in this is that this function is not called in the current socket implementation (for UNIX and INET sockets anyway), and is not instrumented. If it starts getting called then it will have to be instrumented.

pe_enqueueallocfailed is the number of times that we failed to get all of the mbufs required to store a queue of sub-task numbers associated with a socket buffer.

SEE ALSO

getperfstat(2), setsockopt(2), perfd(8)

BUGS

At present *pe_zerodequeues* counts become very high because they can occur in ordinary situations. The code in pe_dequeue should be changed to only increment this counter where a genuine error has occurred.

B.6.2 perf

NAME

perf - control the behaviour of perfd

SYNOPSIS

perf -n
perf -t

DESCRIPTION

perf -n instructs the current perfd process to close the current log file and open a new one, by sending the SIGUSR1 signal to the process.

perf -t instructs the current perfd process to close the current log file and terminate, by sending the SIGTERM signal to the process.

perf finds the pid of the perfd process from **perf_data(5)**.

Note that only the superuser can (successfully) use **perf**.

SEE ALSO

perf_data(5), perfd(8)

B.6.3 perf_warn

NAME

perf_warn - send mail to warn of an abnormal event in the execution of perfd

SYNOPSIS

perf_warn *errname* [*filename*]

DESCRIPTION

perf_warn sends a mail message to RECIPIENT (a variable set within **perf_warn**; root by default) reporting an occurrence of the error *errname*. The value of *errname* must be one of: nospace, hardlim, ebusy, nostart, perfoff, or logpost. See **perfd(8)** for a description of how these abnormal conditions arise.

filename is specified for errors nospace and hardlim, and it is included in the mail message to RECIPIENT.

SEE ALSO

perfd(8)

B.6.4 perfd

NAME

perfd - performance event record logging daemon

SYNOPSIS

perfd [**-b**] [**-d** *logdir*] [**-m** *minfree*] [**-n**] [*event* ...]

DESCRIPTION

perfd is the performance event record logging daemon, which stores event records in a node log file. See **perf.log(5)** for the format of a node log file.

Log files created by **perfd** have names of the form datetime1.datetime2.host, where datetime1 is the GMT date and time at which the log file was opened, datetime2 is the GMT date and time at which the log file was closed, and host is the name of the host on which the file is recorded. While a log file is in use the string "not_terminated" appears as datetime2, with the file being renamed when it is closed.

By default, all events are recorded. The recording of each *event* specified is disabled (see **perfon(2)**). An event can be specified by number or name (see <sys/perform.h> for event names and numbers). The PERF_ prefix is optional when using event names.

perf_warn(8) is executed in a number of situations. One (hardlim) is where the amount of free space on the file system containing the current node log file falls below a certain threshold (see the discussion of the **-m** option). Others are: when event logging has to be terminated because the logging file system is full (nospace); if **perfon(2)** fails to enable event recording (nostart); if a perfd process is already running (ebusy); if event logging has been turned off unexpectedly (perfoff); and if an error occurs while updating the current node log file (logpost).

perf(8) can be used to request that **perfd** close the current node log file and begin another one, and also to ask **perfd** to close the current node log file and terminate.

Note that only the superuser can run **perfd**.

OPTIONS

-b Print debugging messages (which go to stderr).

-d *logdir*
Create node log files in *logdir*. The directory /etc/performance/logfiles is the default.

-m *minfree*
If the amount of free space remaining on the filesystem on which **perfd** is creating node log files drops below *minfree* percent of capacity then **perf_warn(8)** is used to send a mail message containing a warning. The default value of *minfree* is 5.

-n Do not fork. By default **perfd** forks off a child process to run as a daemon. This option is most useful for debugging purposes.

SEE ALSO

perfon(2), perfsvc(2), perf.log(5), perf_data(5), perf(8), perf_warn(8)

Appendix C

A Complete Report from Analyse

This appendix contains the complete report produced by `analyse` for the interaction used for the examples in Chapter 10.

```
******** Overall global level Process stats ********

Statistics for all processes

      Event counts:
                     1  PERF_FORK
                     1  PERF_EXIT
                     1  PERF_ZOMB_EXIT
                     1  PERF_TERM_INPUT_BEGIN
                    14  PERF_RPCCLNT_SEND
                    14  PERF_RPCCLNT_RECV
                    14  PERF_RPC_CALL
                    69  PERF_SWTCH
                    69  PERF_RESUME
                    69  PERF_SETRQ
                    14  PERF_SD_STRATEGY
                    14  PERF_BIOWAIT_END
                    14  PERF_BIOWAIT_START
                     1  PERF_EXEC
                     1  PERF_SIGPAUSE_START
                     1  PERF_SIGPAUSE_END
                    14  PERF_RPCSVC_WAIT
                    14  PERF_RPCCLNT_WAIT
                     1  PERF_TERM_INPUT_END
                    14  PERF_RPCSVC_SEND
                    14  PERF_RPCSVC_RECV

      State counts:
                   239  run
                    69  ready
                     1  created
                     1  zombie
                     1  sleep_child_wait
                    14  sleep_RPC_clnt
                    14  sleep_disc
                     2  sleep_other
```

```
      State Times:
                0.88294  run
                0.33516  ready
                0.00379  created
                0.07998  zombie
                0.48761  sleep_child_wait
                0.35336  sleep_RPC_clnt
                0.64295  sleep_disc
                0.20404  sleep_other

      Disc usage:
            Drive 0
            Access counts:        Part.   Reads  Writes   Total
                                     0       1       0       1
                                     1      12       0      12
                                     6       1       0       1
                                 Total      14       0      14
            Access times:         Part.       Reads     Writes       Total
                                     0     0.03435     0.00000     0.03435
                                     1     0.46873     0.00000     0.46873
                                     6     0.13987     0.00000     0.13987
                                 Total     0.64295     0.00000     0.64295

      RPC Client statistics
            Program 100003, Version 2
                Call counts
                            2  Proc. 1
                            1  Proc. 5
                           11  Proc. 6
                Call times
                      0.02566  Proc. 1
                      0.00998  Proc. 5
                      0.31772  Proc. 6

      RPC Server statistics
            Program 100003, Version 2
                Call counts
                            2  Proc. 1
                            1  Proc. 5
                           11  Proc. 6


******** Overall global level Message stats ********


Statistics for all messages

      Event counts:
                       28  PERF_UDP_RCV
                       28  PERF_UDP_SEND

      State counts:
                       57  queued
                       28  network

      State Times:
                0.13162  queued
                0.25741  network
```

```
******** Critical path global level Process stats ********


Statistics for all processes

      Event counts:
                    1  PERF_FORK
                    1  PERF_EXIT
                    1  PERF_ZOMB_EXIT
                    1  PERF_TERM_INPUT_BEGIN
                   14  PERF_RPCCLNT_SEND
                   14  PERF_RPCCLNT_RECV
                   14  PERF_RPC_CALL
                   52  PERF_SWTCH
                   52  PERF_RESUME
                   52  PERF_SETRQ
                   14  PERF_SD_STRATEGY
                   14  PERF_BIOWAIT_END
                   14  PERF_BIOWAIT_START
                    1  PERF_EXEC
                    1  PERF_SIGPAUSE_END
                    1  PERF_TERM_INPUT_END
                   14  PERF_RPCSVC_SEND
                   14  PERF_RPCSVC_RECV

      State counts:
                  177  run
                   52  ready
                    1  created
                   14  sleep_disc
                    1  sleep_other

      State Times:
                0.62285  run
                0.26833  ready
                0.00379  created
                0.64295  sleep_disc
                0.00991  sleep_other

      Disc usage:
            Drive 0
            Access counts:        Part.    Reads  Writes   Total
                                     0        1       0       1
                                     1       12       0      12
                                     6        1       0       1
                               Total         14       0      14
            Access times:         Part.       Reads     Writes      Total
                                     0     0.03435     0.00000     0.03435
                                     1     0.46873     0.00000     0.46873
                                     6     0.13987     0.00000     0.13987
                               Total       0.64295     0.00000     0.64295

      RPC Client statistics
            Program 100003, Version 2
                Call counts
                            2  Proc. 1
                            1  Proc. 5
                           11  Proc. 6
```

```
      RPC Server statistics
            Program 100003, Version 2
                Call counts
                            2  Proc. 1
                            1  Proc. 5
                           11  Proc. 6


******** Critical path global level Message stats ********


Statistics for all messages

      Event counts:
                   28  PERF_UDP_RCV
                   28  PERF_UDP_SEND

      State counts:
                   57  queued
                   28  network

      State Times:
                0.13162  queued
                0.25741  network


******** Overall node level Process stats ********


Process statistics for all processes on weka (132.181.10.1)

      Event counts:
                   14  PERF_RPCSVC_WAIT
                   14  PERF_RPCSVC_SEND
                   14  PERF_RPCSVC_RECV

      State counts:
                   28  run

      State Times:
                0.24995  run

      Disc usage:

      RPC Client statistics

      RPC Server statistics
            Program 100003, Version 2
                Call counts
                            2  Proc. 1
                            1  Proc. 5
                           11  Proc. 6
```

```
Process statistics for all processes on tui (132.181.10.4)

      Event counts:
                    1  PERF_FORK
                    1  PERF_EXIT
                    1  PERF_ZOMB_EXIT
                    1  PERF_TERM_INPUT_BEGIN
                   14  PERF_RPCCLNT_SEND
                   14  PERF_RPCCLNT_RECV
                   14  PERF_RPC_CALL
                   69  PERF_SWTCH
                   69  PERF_RESUME
                   69  PERF_SETRQ
                   14  PERF_SD_STRATEGY
                   14  PERF_BIOWAIT_END
                   14  PERF_BIOWAIT_START
                    1  PERF_EXEC
                    1  PERF_SIGPAUSE_START
                    1  PERF_SIGPAUSE_END
                   14  PERF_RPCCLNT_WAIT
                    1  PERF_TERM_INPUT_END

      State counts:
                  211  run
                   69  ready
                    1  created
                    1  zombie
                    1  sleep_child_wait
                   14  sleep_RPC_clnt
                   14  sleep_disc
                    2  sleep_other

      State Times:
                0.63299  run
                0.33516  ready
                0.00379  created
                0.07998  zombie
                0.48761  sleep_child_wait
                0.35336  sleep_RPC_clnt
                0.64295  sleep_disc
                0.20404  sleep_other

      Disc usage:
            Drive 0
            Access counts:        Part.   Reads  Writes   Total
                                     0       1       0       1
                                     1      12       0      12
                                     6       1       0       1
                                 Total      14       0      14
            Access times:         Part.       Reads      Writes       Total
                                     0     0.03435     0.00000     0.03435
                                     1     0.46873     0.00000     0.46873
                                     6     0.13987     0.00000     0.13987
                                 Total     0.64295     0.00000     0.64295
```

```
      RPC Client statistics
            Program 100003, Version 2
                Call counts
                            2  Proc. 1
                            1  Proc. 5
                           11  Proc. 6
                Call times
                      0.02566  Proc. 1
                      0.00998  Proc. 5
                      0.31772  Proc. 6

      RPC Server statistics


******** Overall node level Message stats ********


Statistics for messages between processes on tui (132.181.10.4), and
    processes on weka (132.181.10.1)

      Event counts:
                   14  PERF_UDP_RCV
                   14  PERF_UDP_SEND

      State counts:
                   28  queued
                   14  network

      State Times:
                0.06916  queued
                0.04033  network


Statistics for messages between processes on tui (132.181.10.4), and
    processes on tui (132.181.10.4)

      Event counts:

      State counts:
                    1  queued

      State Times:
                0.01446  queued


Statistics for messages between processes on weka (132.181.10.1), and
    processes on tui (132.181.10.4)

      Event counts:
                   14  PERF_UDP_RCV
                   14  PERF_UDP_SEND

      State counts:
                   28  queued
                   14  network

      State Times:
                0.04800  queued
                0.21708  network
```

```
******** Critical path node level Process stats ********


Process statistics for all processes on weka (132.181.10.1)

      Event counts:
                  14  PERF_RPCSVC_SEND
                  14  PERF_RPCSVC_RECV

      State counts:
                  14  run

      State Times:
             0.05483  run

      Disc usage:

      RPC Client statistics

      RPC Server statistics
            Program 100003, Version 2
                Call counts
                            2  Proc. 1
                            1  Proc. 5
                           11  Proc. 6


Process statistics for all processes on tui (132.181.10.4)

      Event counts:
                   1  PERF_FORK
                   1  PERF_EXIT
                   1  PERF_ZOMB_EXIT
                   1  PERF_TERM_INPUT_BEGIN
                  14  PERF_RPCCLNT_SEND
                  14  PERF_RPCCLNT_RECV
                  14  PERF_RPC_CALL
                  52  PERF_SWTCH
                  52  PERF_RESUME
                  52  PERF_SETRQ
                  14  PERF_SD_STRATEGY
                  14  PERF_BIOWAIT_END
                  14  PERF_BIOWAIT_START
                   1  PERF_EXEC
                   1  PERF_SIGPAUSE_END
                   1  PERF_TERM_INPUT_END

      State counts:
                 163  run
                  52  ready
                   1  created
                  14  sleep_disc
                   1  sleep_other

      State Times:
             0.56802  run
             0.26833  ready
             0.00379  created
             0.64295  sleep_disc
             0.00991  sleep_other
```

```
      Disc usage:
            Drive 0
            Access counts:        Part.    Reads  Writes   Total
                                     0        1       0       1
                                     1       12       0      12
                                     6        1       0       1
                                 Total       14       0      14
            Access times:         Part.       Reads     Writes       Total
                                     0     0.03435     0.00000     0.03435
                                     1     0.46873     0.00000     0.46873
                                     6     0.13987     0.00000     0.13987
                                 Total     0.64295     0.00000     0.64295

      RPC Client statistics
            Program 100003, Version 2
                Call counts
                             2  Proc. 1
                             1  Proc. 5
                            11  Proc. 6

      RPC Server statistics


******** Critical path node level Message stats ********


Statistics for messages between processes on tui (132.181.10.4), and
    processes on weka (132.181.10.1)

      Event counts:
                    14  PERF_UDP_RCV
                    14  PERF_UDP_SEND

      State counts:
                    28  queued
                    14  network

      State Times:
                0.06916  queued
                0.04033  network


Statistics for messages between processes on tui (132.181.10.4), and
    processes on tui (132.181.10.4)

      Event counts:

      State counts:
                     1  queued

      State Times:
                0.01446  queued
```

```
Statistics for messages between processes on weka (132.181.10.1), and
    processes on tui (132.181.10.4)

      Event counts:
                  14  PERF_UDP_RCV
                  14  PERF_UDP_SEND

      State counts:
                  28  queued
                  14  network

      State Times:
             0.04800  queued
             0.21708  network


******** Overall process level Process stats ********


Process statistics for process 322 /bin/ls on tui (132.181.10.4)

      Event counts:
                   1  PERF_EXIT
                   6  PERF_RPCCLNT_SEND
                   6  PERF_RPCCLNT_RECV
                   6  PERF_RPC_CALL
                  17  PERF_SWTCH
                  18  PERF_RESUME
                  18  PERF_SETRQ
                   2  PERF_SD_STRATEGY
                   2  PERF_BIOWAIT_END
                   2  PERF_BIOWAIT_START
                   1  PERF_EXEC
                   6  PERF_RPCCLNT_WAIT

      State counts:
                  57  run
                  18  ready
                   1  created
                   1  zombie
                   6  sleep_RPC_clnt
                   2  sleep_disc
                   1  sleep_other

      State Times:
             0.34171  run
             0.08259  ready
             0.00379  created
             0.07998  zombie
             0.11034  sleep_RPC_clnt
             0.17422  sleep_disc
             0.00991  sleep_other
```

```
Disc usage:
      Drive 0
      Access counts:        Part.    Reads  Writes    Total
                               0        1       0        1
                               6        1       0        1
                           Total        2       0        2
      Access times:         Part.       Reads     Writes      Total
                               0     0.03435    0.00000    0.03435
                               6     0.13987    0.00000    0.13987
                           Total     0.17422    0.00000    0.17422

RPC Client statistics
      Program 100003, Version 2
          Call counts
                       2  Proc. 1
                       1  Proc. 5
                       3  Proc. 6
          Call times
                 0.02566  Proc. 1
                 0.00998  Proc. 5
                 0.07470  Proc. 6

RPC Server statistics

Process validation
      Quantity    Monitor              should be    Rusage
      CPU run time          0.34171 = (approx)       0.32000
      vol cswtch            9 = (exact)              9
      invol cswtch                8 = (exact)                8
      Msg recvs             0 = (exact)              0
      Msg sends             0 <=                     0
      Swaps                 0 = (exact)              0


Process statistics for process 151  on tui (132.181.10.4)

      Event counts:
                     1  PERF_FORK
                     1  PERF_ZOMB_EXIT
                     1  PERF_TERM_INPUT_BEGIN
                     8  PERF_RPCCLNT_SEND
                     8  PERF_RPCCLNT_RECV
                     8  PERF_RPC_CALL
                    52  PERF_SWTCH
                    51  PERF_RESUME
                    51  PERF_SETRQ
                    12  PERF_SD_STRATEGY
                    12  PERF_BIOWAIT_END
                    12  PERF_BIOWAIT_START
                     1  PERF_SIGPAUSE_START
                     1  PERF_SIGPAUSE_END
                     8  PERF_RPCCLNT_WAIT
                     1  PERF_TERM_INPUT_END

      State counts:
                   154  run
                    51  ready
                     1  sleep_child_wait
                     8  sleep_RPC_clnt
                    12  sleep_disc
                     1  sleep_other
```

```
      State Times:
                0.29128  run
                0.25257  ready
                0.48761  sleep_child_wait
                0.24302  sleep_RPC_clnt
                0.46873  sleep_disc
                0.19413  sleep_other

      Disc usage:
            Drive 0
            Access counts:        Part.    Reads  Writes   Total
                                     1       12       0      12
                                 Total       12       0      12
            Access times:         Part.       Reads     Writes      Total
                                     1     0.46873    0.00000    0.46873
                                 Total     0.46873    0.00000    0.46873

      RPC Client statistics
            Program 100003, Version 2
                Call counts
                            8  Proc. 6
                Call times
                      0.24302  Proc. 6

      RPC Server statistics


Process statistics for process 104  on weka (132.181.10.1)

      Event counts:
                     14  PERF_RPCSVC_WAIT
                     14  PERF_RPCSVC_SEND
                     14  PERF_RPCSVC_RECV

      State counts:
                     28  run

      State Times:
                0.24995  run

      Disc usage:

      RPC Client statistics

      RPC Server statistics
            Program 100003, Version 2
                Call counts
                            2  Proc. 1
                            1  Proc. 5
                           11  Proc. 6


******** Overall process level Message stats ********


Statistics for messages between process 151 on tui (132.181.10.4), and
    process 104 on weka (132.181.10.1)

      Event counts:
                      8  PERF_UDP_RCV
                      8  PERF_UDP_SEND
```

```
      State counts:
                  16  queued
                   8  network

      State Times:
              0.03655  queued
              0.02389  network


Statistics for messages between process 104 on weka (132.181.10.1), and
    process 322 on tui (132.181.10.4)

      Event counts:
                   6  PERF_UDP_RCV
                   6  PERF_UDP_SEND

      State counts:
                  12  queued
                   6  network

      State Times:
              0.02609  queued
              0.05037  network


Statistics for messages between process 322 on tui (132.181.10.4), and
    process 151 on tui (132.181.10.4)

      Event counts:

      State counts:
                   1  queued

      State Times:
              0.01446  queued


Statistics for messages between process 322 on tui (132.181.10.4), and
    process 104 on weka (132.181.10.1)

      Event counts:
                   6  PERF_UDP_RCV
                   6  PERF_UDP_SEND

      State counts:
                  12  queued
                   6  network

      State Times:
              0.03261  queued
              0.01644  network


Statistics for messages between process 104 on weka (132.181.10.1), and
    process 151 on tui (132.181.10.4)

      Event counts:
                   8  PERF_UDP_RCV
                   8  PERF_UDP_SEND
```

```
      State counts:
                   16  queued
                    8  network

      State Times:
                0.02191  queued
                0.16671  network


******** Critical path process level Process stats ********


Process statistics for process 322 /bin/ls on tui (132.181.10.4)

      Event counts:
                    1  PERF_EXIT
                    6  PERF_RPCCLNT_SEND
                    6  PERF_RPCCLNT_RECV
                    6  PERF_RPC_CALL
                   11  PERF_SWTCH
                   12  PERF_RESUME
                   12  PERF_SETRQ
                    2  PERF_SD_STRATEGY
                    2  PERF_BIOWAIT_END
                    2  PERF_BIOWAIT_START
                    1  PERF_EXEC

      State counts:
                   39  run
                   12  ready
                    1  created
                    2  sleep_disc
                    1  sleep_other

      State Times:
                0.31973  run
                0.06773  ready
                0.00379  created
                0.17422  sleep_disc
                0.00991  sleep_other

      Disc usage:
            Drive 0
            Access counts:        Part.   Reads  Writes   Total
                                     0       1       0       1
                                     6       1       0       1
                                 Total       2       0       2
            Access times:         Part.       Reads      Writes       Total
                                     0     0.03435     0.00000     0.03435
                                     6     0.13987     0.00000     0.13987
                                 Total     0.17422     0.00000     0.17422

      RPC Client statistics
            Program 100003, Version 2
                Call counts
                            2  Proc. 1
                            1  Proc. 5
                            3  Proc. 6

      RPC Server statistics
```

```
Process statistics for process 151  on tui (132.181.10.4)

      Event counts:
                    1  PERF_FORK
                    1  PERF_ZOMB_EXIT
                    1  PERF_TERM_INPUT_BEGIN
                    8  PERF_RPCCLNT_SEND
                    8  PERF_RPCCLNT_RECV
                    8  PERF_RPC_CALL
                   41  PERF_SWTCH
                   40  PERF_RESUME
                   40  PERF_SETRQ
                   12  PERF_SD_STRATEGY
                   12  PERF_BIOWAIT_END
                   12  PERF_BIOWAIT_START
                    1  PERF_SIGPAUSE_END
                    1  PERF_TERM_INPUT_END

      State counts:
                  124  run
                   40  ready
                   12  sleep_disc

      State Times:
              0.24829  run
              0.20060  ready
              0.46873  sleep_disc

      Disc usage:
            Drive 0
            Access counts:        Part.    Reads  Writes   Total
                                     1       12       0      12
                                 Total       12       0      12
            Access times:         Part.       Reads     Writes      Total
                                     1     0.46873    0.00000    0.46873
                                 Total     0.46873    0.00000    0.46873

      RPC Client statistics
            Program 100003, Version 2
                Call counts
                            8  Proc. 6

      RPC Server statistics


Process statistics for process 104  on weka (132.181.10.1)

      Event counts:
                   14  PERF_RPCSVC_SEND
                   14  PERF_RPCSVC_RECV

      State counts:
                   14  run

      State Times:
              0.05483  run

      Disc usage:

      RPC Client statistics
```

```
      RPC Server statistics
            Program 100003, Version 2
                Call counts
                            2  Proc. 1
                            1  Proc. 5
                           11  Proc. 6


******** Critical path process level Message stats ********


Statistics for messages between process 151 on tui (132.181.10.4), and
    process 104 on weka (132.181.10.1)

      Event counts:
                      8  PERF_UDP_RCV
                      8  PERF_UDP_SEND

      State counts:
                     16  queued
                      8  network

      State Times:
                0.03655  queued
                0.02389  network


Statistics for messages between process 104 on weka (132.181.10.1), and
    process 322 on tui (132.181.10.4)

      Event counts:
                      6  PERF_UDP_RCV
                      6  PERF_UDP_SEND

      State counts:
                     12  queued
                      6  network

      State Times:
                0.02609  queued
                0.05037  network


Statistics for messages between process 322 on tui (132.181.10.4), and
    process 151 on tui (132.181.10.4)

      Event counts:

      State counts:
                      1  queued

      State Times:
                0.01446  queued


Statistics for messages between process 322 on tui (132.181.10.4), and
    process 104 on weka (132.181.10.1)

      Event counts:
                      6  PERF_UDP_RCV
                      6  PERF_UDP_SEND
```

```
      State counts:
                   12  queued
                    6  network

      State Times:
              0.03261  queued
              0.01644  network


Statistics for messages between process 104 on weka (132.181.10.1), and
    process 151 on tui (132.181.10.4)

      Event counts:
                    8  PERF_UDP_RCV
                    8  PERF_UDP_SEND

      State counts:
                   16  queued
                    8  network

      State Times:
              0.02191  queued
              0.16671  network
```

Acknowledgements

I wish to acknowledge the contributions of others to this work. Professor John Penny has supervised this work. Throughout, John has been a source of enthusiastic support, and he has made a substantial contribution to the preparation of this thesis. Fellow PhD students Greg Ewing and Richard Pascoe took part in many discussions of this work during research meetings.

The Department of Computer Science has also assisted by providing a good environment in which to carry out this work.

Finally, I wish to acknowledge the support of my family. My parents, Dot and Jim, gave me every opportunity to get a good education, and for that I am very grateful. Also, thanks go to Trudy (whom I married during this work), who has been denied many hours of my time, as have Thomas and Edward, who were both born during this work, and who have both grown an alarming amount since it started.

References

[ABRA77] Abrams, M. D., Techniques for evaluating the effectiveness of interactive computing service, *Proc 1977 Annual Conf. of the Association for Computing Machinery*, Seattle, Washington, 1977.

[ALME85] Almes, G. T., Black, A. P., Lazowska, E. D., and Noe, J. D., The Eden System: A Technical Review, *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 1, January 1985, pp 43-59.

[ANDE90] Anderson, T. E., and Lazowska, E. D., Quartz: A Tool for Tuning Parallel Program Performance, *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1990, pp 115-125.

[ANDR83] Andrews, G. R, and Schneider, F. B., Concepts and Notations for Concurrent Programming, *ACM Computing Surveys*, Vol. 15, No. 1, March 1983, pp 3-43.

[ANDR89] Andrews, M., Chadwick, D., Chelluri, S., DiPirro, A., Flynn, T., Rybak, E., Traister, L., and Wyse, N., *UNIX International (UI) White Paper on Performance Management Facilities*, Unix International, December 1989.

[ARAL88] Aral, Z., and Gertner, I., Non-Intrusive and Interactive Profiling in Parasight, *Proceedings of the 1988 ACM/SIGPLAN conference on PPEALS*, SIGPLAN Notices, Vol. 23, No. 9, September 1988, pp 21-30.

[ASHT84] Ashton, P. J., *ASHMON - A Monitoring Package for PRIME 50 Series Machines*, Department of Computer Science, University of Canterbury, 1984.

[ASHT90] Ashton, P. J., and Penny, J. P., *The Modelling and Performance Analysis of User Interactions with a Distributed System*, Technical Report COSC 06/90, Department of Computer Science, University of Canterbury, 1990.

[ASHT91a] Ashton, P. J., and Penny, J. P., Decomposition of Interactive Response Times for Loosely-Coupled Distributed Systems, *Proceedings of the 14th Australian Computer Science Conference*, Sydney, February 1991, pp 19-1 - 19-10.

[ASHT91b] Ashton, P. J., and Penny, J. P., *The Interaction Network: a Performance Evaluation Tool for Loosely-Coupled Distributed Systems*, Technical Report COSC 01/91, Department of Computer Science, University of Canterbury, 1991.

[ASHT92] Ashton, P. J., and Penny, J. P., Experiments with an Algorithm for High-Resolution Clock Synchronisation, *Proceedings of the 15th Australian Computer Science Conference*, Hobart, January 1992, pp 41-55.

[ASTR84] *The Astronomical Almanac for 1984*, Washington, London, 1984.

[BACH86] Bach, M. J., *The Design of the UNIX Operating System,* Prentice-Hall, Englewood Cliffs, NJ, 1986.

[BAL89] Bal, H. E., Steiner, J. G., and Tanenbaum, A. S., Programming Languages for Distributed Computing Systems, *ACM Computing Surveys*, Vol. 21, No. 3, September 1989, pp 261-322.

[BALL89] Ball, C. R., Leung, T. W., and Waldspurger, C. A., Analyzing Patterns of Message Passing, *Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, San Diego, September 1988, SIGPLAN Notices, Vol. 24, No. 4, April 1989, pp 191-193.

[BARN89] Barnes, G., *How To Use Xgrab (Version 2.3)*, Tera Computer Company, Seattle, Wa., 1989.

[BENB84] Benbasat, I., and Wand, Y., A structured approach to designing human-computer dialogues, *International Journal of Man-Machine Studies*, Vol. 21, 1984, pp 105-126.

[BERK85] Csh command, *UNIX Programmer's Manual (Section 2)*, 4th Berkeley UNIX Distribution Release 3, February 1985.

[BERS91] Bershad, B. N., Anderson, T. E., Lazowska, E. D., and Levy, H. M., User-Level Interprocess Communication for Shared Memory Multiprocessors, *ACM Transactions on Computer Systems*, Vol. 9, No. 2, May 1991, pp 175-198.

[BIRR84] Birrell, A. D., and Nelson, B. J., Implementing Remote Procedure Calls, *ACM Transactions on Computer Systems*, Vol. 2, No. 1, February 1984, pp 39-59.

[BRAN89] Brantley, W. C., Brochard, L. G., Bolmarcich, A., Chang, H. Y., McAuliffe, K. P., and Ngo, T. A., Initial Experiences with RP3 Performance Monitoring, *International Journal of High Speed Computing*, Vol. 1, No. 4, 1989, pp 543-561.

[CALI67] Calingaert, P., System Performance Evaluation: Survey and Appraisal, *Communications of the ACM*, Vol. 10, No. 1, 1967, pp 12-18.

[CARL88] de Carlini, U., and Villano, U., A Simple Algorithm for Clock Synchronization in Transputer Networks, *Software–Practice and Experience*, Vol. 18, No. 4, April 1988, pp 331-347.

[CHER84] Cheriton, D. R., The V Kernel: A Software Base for Distributed Systems, *IEEE Software*, Vol. 1, No. 2, April 1984, pp 19-42.

[CHER85] Cheriton, D. R., and Zwaenepoel, W., Distributed Process Groups in the V Kernel, *ACM Transactions on Computer Systems*, Vol. 3, No. 2, May 1985, pp 77-107.

[COME88] Comer, D., *Internetworking with TCP/IP: Principles, Protocols and Architecture*, Prentice-Hall, Englewood Cliffs, NJ, 1988.

[COUL88] Colouris, G. F., and Dollimore, J., *Distributed Systems: Concepts and Design,* Addison-Wesley, 1988.

[CRIS89] Cristian, F., Probabalistic Clock Synchronization, *Distributed Computing*, Vol. 3, No. 3, 1989, pp 146-158.

[DANZ90] Danzig, P. B., and Melvin, S., High Resolution timing with low resolution clocks and a microsecond timer for Sun workstations, *Operating Systems Review*, Vol. 24, No. 1, January 1990, pp 23-26.

[DANZ91] Danzig, P. B., An Analytical Model of Operating System Protocol Processing Including Effects of Multiprogramming, *Proceedings of the 1991 ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, May 1991, pp 11-20.

[DEIT84] Deitel, H. M., *An Introduction to Operating Systems,* Revised 1st edition, Addison-Wesley, 1984.

[DUCH89] Duchamp, D., Analysis of Transaction Management Performance, *Proceedings of the Twelfth ACM Symposium on Operating System Principles*, December 1989, pp 177-190.

[DUDA87] Duda, A., Harrus, G., Haddad, Y., and Bernard, G., Estimating Global Time in Distributed Systems, *7th International Conference on Distributed Computing Systems*, September 1987, pp 299-306.

[ELLI73] Ellingson, C. E., and Kulpinski, R. J., Dissemination of System Time, *IEEE Transactions on Communications*, Vol. COM-21, No. 5, May 1973, pp 605-623.

[EVEN79] Even, S., *Graph Algorithms*, Computer Science Press, Rockville, Maryland, 1979.

[FELD79] Feldman, S. I., MAKE - a program for maintaining computer programs, *Software–Practice and Experience*, Vol. 9, No. 4, April 1979, pp 255-265.

[FERR78] Ferrari, D., *Computer Systems Performance Evaluation,* Prentice-Hall, Englewood Cliffs, NJ, 1978.

[FERR83] Ferrari, D., Serazzi, G., and Zeigner, A., *Measurement and Tuning of Computer Systems,* Prentice-Hall, Englewood Cliffs, NJ, 1983.

[FOLE82] Foley, J. D., and van Dam, A., *Fundamentals of Interactive Computer Graphics,* Prentice-Hall, Englewood Cliffs, NJ, 1982.

[FORT88] Fortier, Paul J., *Design of Distributed Operating Systems*, McGraw-Hill, Singapore, 1988.

[GOSC91] Goscinski, A., *Distributed Operating Systems: The Logical Design*, Addison-Wesley, Sydney, 1991.

[GRAH82] Graham, S. L., Kessler, P. B., and McKusick, M. K., gprof: A Call Graph Execution Profiler, *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, SIGPLAN Notices, Vol. 17, No. 6, June 1982, pp 120-126.

[GRAY89] Gray, C. G., and Cheriton, D. R., Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency, *Proceedings of the Twelfth ACM Symposium on Operating System Principles*, December 1989, pp 202-210.

[GUER83] Guerich, W., and Mertens, B., Comparison of internal and external interactive response times, *Computer Performance*, Vol. 4, No. 1, March 1983, pp 13-20.

[GUSE89] Gusella, R., and Zatti, S., The Accuracy of the Clock Synchronization Achieved by TEMPO in Berkeley UNIX 4.3BSD, *IEEE Transactions on Software Engineering*, Vol. 15, No. 7, July 1989, pp 847-853.

[HABA90] Haban, D., and Wybranietz, D., A Hybrid Monitor for Behaviour and Performance Analysis of Distributed Systems, *IEEE Transactions on Software Engineering*, Vol. 16, No. 2, February 1990, pp 197-211.

[HART89] Hartson, H. R., and Hix, D., Human-Computer Interface Development: Concepts and Systems for Its Management, *ACM Computing Surveys*, Vol. 21, No. 1, March 1989, pp 5-92.

[HEID84] Heidelberger, P., and Lavenberg, S. S., Computer Performance Evaluation Methodology, *IEEE Transactions on Computers*, Vol. C-33, No. 12, December 1984, pp 1195-1220.

[HOLL91a] Hollingsworth, J., Irvin, B., and Miller, B. P., *IPS User's Guide, Version 4.0*, Computer Science Department, University of Wisconsin-Madison, June 1991.

[HOLL91b] Hollingsworth, J., Irvin, R. B., and Miller, B. P., The Integration of Application and System Based Metrics in A Parallel Program Performance Tool, *Proc. of the 1991 ACM SIGPLAN Notices Symposium on Principals and Practice of Parallel Programming*, April 1991.

[HOUG89] Hough, A. A., and Cuny, J. E., Initial Experiences with a Pattern-Oriented Parallel Debugger, *SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, Madison Wi, May 1988, SIGPLAN Notices, Vol. 24, No. 1, January 1989, pp 195-205.

[JAIN91] Jain, R., *The Art of Computer Systems Performance Analysis*, John Wiley & Sons, 1991.

[JOYC87] Joyce, J., Lomow, G., Slind, K., and Unger, B., Monitoring Distributed Systems, *ACM Transaction on Computer Systems*, Vol. 5, No.2, May 1987, pp 121-150.

[JUL88] Jul, E., Levy, H., Hutchinson, N., and Black, A., Fine-Grained Mobility in the Emerald System, *Transactions on Computer Systems*, Vol. 6, No. 1, February 1988, pp 109-133.

[KERO87] Kerola, T., and Schwetman, H., Monit: A Performance Monitoring Tool for Parallel and Pseudo-Parallel Programs, *Proceedings of the 1987 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1987, pp 163-174.

[KOHL91] Kohl, J. T., The Evolution of the Kerberos Authentication Service, *Proceedings of the Spring 1991 EurOpen Conference*, Tromsø, Norway, May 1991, pp 295-313.

[KOPE87] Kopetz, H., and Ochsenreiter, W., Interval Measurement in Distributed Real Time Systems, *7th International Conference on Distributed Computing Systems*, September 1987, pp 292-298.

[KRIS85] Krishna, C. M., Shin, K. G., and Butler, R. W., Ensuring fault tolerance of phase-locked clocks, *IEEE Transactions on Computers*, Vol. C-34, No. 8, August 1985, pp 752-756.

[KUND86] Kundu, S., The Call-Return Tree and Its Application to Program Performance Analysis, *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 11, November 1986, pp 1096-1098.

[LAMP78] Lamport, L., Time, clocks, and the ordering of events in distributed systems, *Communications of the ACM*, Vol. 21, No. 7, July 1978, pp 558-565.

[LAMP85] Lamport, L., and Melliar-Smith, P. M., Synchronizing Clocks in the Presence of Faults, *Journal of the ACM*, Vol. 32, No. 1, January 1985, pp 52-78.

[LEFF88] Leffler, S. J., McKusick, M. K., Karels, M. J., and Quaterman, J. S., *The Design and Implementation of the 4.3 BSD UNIX Operating System,* Addison-Wesley, 1988.

[LEUN88] Leung, C. H. C., *Quantitative Analysis of Computer Systems,* John Wiley & Sons, 1988.

[MAHJ84] Mahjoub, A., On the Static Evaluation of Distributed Systems Performance, *The Computer Journal*, Vol. 27, No. 3, 1984, pp 201-208.

[MAND89] Mandell, D., and Trease, H., Parallel Processing a Real Code–A Case History, in *Instrumentation for Future Parallel Computing Systems*, Simmons, M., Koskela, R., and Bucher, I. eds., Addison-Wesley, 1989, pp 209-222.

[MASO87] Mason, W. A., Distributed Processing: The State of the Art, *Byte*, Vol. 12, No. 13, November 1987, pp 291-297.

[MCDO87] McDonell, K. J., Taking Performance Evaluation Out of the ``Stone Age'', *Proceedings Summer Usenix Technical Conference*, Phoenix, Arizona, June 1987, pp 407-417.

[MILL86] Miller, B. P., Macrander, C. M., and Sechrest, S., A Distributed Programs Monitor for Berkeley Unix, *Software–Practice and Experience*, Vol. 16, No. 2, February 1986, pp 183-200.

[MILL87] Miller, B. P., and Yang, C.-Q., IPS: An Interactive and Automatic Performance Measurement Tool for Parallel and Distributed Programs, *Proceedings of the 7th International Conference on Distributed Computing Systems*, September 1987, pp 482-489.

[MILL88a] Miller, B. P., DPM: A Measurement System for Distributed Programs, *IEEE Transactions on Computers*, Vol. 37, No. 2, February 1988, pp 243-248.

[MILL88b] Mills, D. L., *Network Time Protocol (Version 1) Specification and Implementation*, DARPA Network Working Group Report RFC-1059, University of Delaware, July 1988.

[MILL89a] Mills, D. L., *Network Time Protocol (Version 2) Specification and Implementation*, DARPA Network Working Group Report RFC-1119, University of Delaware, September 1989.

[MILL89b] Miller, B. P., and Choi, J.-D., A Mechanism for Efficient Debugging of Parallel Programs, *SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, Madison Wi May, 1988, SIGPLAN Notices, Vol. 24, No. 1, 1989, pp 141-150.

[MILL90a] Miller, B. P., Clark, M., Hollingsworth, J., Kierstead, S., Lim, S.-S., and Torzewski, T., IPS-2: The Second Generation of a Parallel Program Measurement System, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 2, April 1990, pp 206-217.

[MILL90b] Mills, D. L., On the Accuracy and Stability of Clocks Synchronized by the Network Time Protocol in the Internet System, *Computer Communication Review*, Vol. 20, No. 1, January 1990, pp 65-75.

[MUKK88] Mukkamala, R., Bruell, S. C., and Shultz, R. K., Design of Partially Replicated Distributed Database Systems: An Integrated Methodology, *Proceedings of the 1988 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1988, pp 187-196.

[MULL86] Mullender, S. J., and Tanenbaum, A. S., The Design of a Capability-Based Distributed Operating System, *The Computer Journal*, Vol. 29, No. 4, March 1986, pp 289-300.

[NIEL86] Nielsen, J., A Virtual protocol model for computer-human interaction, *International Journal of Man-Machine Studies*, Vol. 24, 1986, pp 301-312.

[PENN84] Penny, J. P., and Ashton, P. J., Measurement and description of time-sharing system response, *Computer Performance*, Vol. 5, No. 3, September 1984, pp 144-152.

[PENN86] Penny, J. P., Ashton, P. J., and Wilkinson, A. L., Data Recording and Monitoring for Analysis of System Response Times, *Computer Journal*, Vol. 29, No. 5, 1986, pp 396-403.

[PENN88] Penny, J. P., Ashton, P. J., and Tripp, D., Instrumenting Systems to Measure Components of Interactive Response Times, *Australian Computer Journal*, Vol. 20, No. 2, May 1988, pp 79-84.

[POST80] Postel, J., *User Datagram Protocol*, USC/Information Sciences Institute Rep. RFC 768, August 1980.

[POST81a] Postel, J., Ed., *Internet Protocol*, USC/Information Sciences Institute Rep. RFC 791, September 1981.

[POST81b] Postel, J., *Internet Control Message Protocol*, USC/Information Sciences Institute Rep. RFC 792, September 1981.

[POST81c] Postel, J., Ed., *Transmission Control Protocol*, USC/Information Sciences Institute Rep. RFC 793, September 1981.

[POST83a] Postel, J., *Daytime Protocol*, USC/Information Sciences Institute Rep. RFC 867, May 1983.

[POST83b] Postel, J and Harrenstein, K., *Time Protocol*, USC/Information Sciences Institute Rep. RFC 868, May 1983.

[QUAR85] Quarterman, J. S., Silberschatz, A., and Peterson, J. L., 4.2BSD and 4.3BSD as Examples of the UNIX Sysetm, *ACM Computing Surveys*, Vol. 17, No. 4, December 1985, pp 379-418.

[RAMA90a] Ramanathan, P., Kandlur, D. D., and Shin, K. G, Hardware-Assisted Software Clock Synchronization for Homogeneous Distributed Systems, *IEEE Transactions on Computers*, Vol. 39, No. 4, April 1990, pp 514-524.

[RAMA90b] Ramanathan, P., Shin, K. G., and Butler, R. W., Fault-Tolerant Clock Synchronization in Distributed Systems, *Computer*, Vol. 23, No. 10, October 1990, pp 33-42.

[REED89] Reed, D. A., and Rudolph, D. C., Experiences with Hypercube Operating System Instrumentation, *International Journal of High Speed Computing*, Vol. 1, No. 4, December 1989, pp 517-542.

[RENE89] van Renesse, R., *The Function Processing Model*, PhD thesis, Vrije University, Amsterdam, 1989.

[RITC74] Ritchie, D. M., and Thompson, K., The UNIX Time-Sharing System, *Communications ot the ACM*, Vol. 17, No. 7, July 1974, pp 365-375.

[ROWE87] Rowe, L., A., Davis, M., Messinger, E., Meyer, C., Spirakis, C., and Tuan, A., A Browser for Directed Graphs, *Software–Practice and Experience*, Vol. 17, No. 1, January 1987, pp 61-76.

[SAND85] Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., and Lyon, B., Design and Implementation of the Sun Network Filesystem, *Proceedings of the 1985 Summer Usenix Conference*, June 1985, pp 119-130.

[SCHE90] Scheifler, R., and Gettys, R., *X Window System*, Second Edition, Digital Press, 1990.

[SHNE84] Shneiderman, B., Response Time and Display Rate in Human Performance with Computers, *ACM Computing Surveys*, Vol. 16, No. 3, September 1984, pp 265-285.

[SILB91] Silberschatz, A., Peterson, J. L., and Galvin, P. B., *Operating System Concepts*, Third edition, Addison-Wesley, Reading, Ma, 1991.

[SMIT88] Smith, J. M., A Survey of Process Migration Mechanisms, *Operating Systems Review*, Vol. 22, No. 3, July 1988, pp 28-40.

[SNOD86] Snodgrass, R., and Ahn, I., Temporal Databases, *Computer*, Vol. 19, No. 9, September 1986, pp 35-42.

[SNOD87] Snodgrass, R., The Temporal Query Language TQuel, *ACM Transactions on Database Systems*, Vol. 12, No. 2, June 1987, pp 247-298.

[SNOD88] Snodgrass, R., A Relational Approach to Monitoring Complex Systems, *ACM Transactions on Computer Systems*, Vol. 6, No. 2, May 1988, pp 157-196.

[SOMI91] Somin, Y., Performance Analysis in GUI Environments under UNIX, *Proceedins of the 1991 CMG Conference*, Computer Measurement Group, 1991.

[SUN88a] *System Services Overview (Revision A of 9 May 1988)*, Sun Microsystems, 1988.

[SUN88b] *Network Programming (Revision A of 9 May 1988)*, Sun Microsystems, 1988.

[SUN88c] *Security Features Guide (Revision A of 9 May 1988)*, Sun Microsystems, 1988.

[SUN88d] *Writing Device Drivers (Revision A of 9 May 1988)*, Sun Microsystems, 1988.

[TAM90] Tam, M.-C., Smith, J. M., and Farber, D. J., A Taxonomy-Based Comparison of Several Distributed Shared Memory Systems, *Operating Systems Review*, Vol. 24, No. 3, July 1990, pp 40-67.

[TANE85] Tanenbaum, A. S., and van Renesse, R., Distributed Operating Systems, *ACM Computing Surveys*, Vol. 17, No. 4, December 1985, pp 419-470.

[TANE89] Tanenbaum, A. S., van Renesse, R., van Staveren, H., Sharp, G. J., Mullender, S. J., Jansen, A. J., and van Rossum, G., *Experiences with the Ameoba Distributed Operating System*, Report IR-194, Dept. of Mathematics and Computer Science, Vrije University, July 1989.

[TETZ79] Tetzlaff, W. H., State Sampling of Interactive VM/370 Users, *IBM Systems Journal*, Vol. 18, No. 1, 1979, pp 164-180.

[TETZ82] Tetzlaff, W., and Beretvas, T., A New Approach to VM performance Analysis, *Proceedings of the 1982 CMG Conference*, Computer Measurement Group, 1982, pp 339-350.

[TEVA89] Tevanian, A. Jr., and Smith, B., Mach: The Model for Future Unix, *Byte*, Vol. 14, No. 12, November 1989, pp 411-416.

[THOO91] Thoo, L. C., *IN-browser: A Graphical Browser for Interaction Networks*, Department of Computer Science, University of Canterbury, October 1991.

[TSAI90] Tsai, J. J. P., Fang, K.-Y., Chen, H.-Y., and Bi, Y.-D., A Noninterference Monitoring and Replay Mechanism for Real-Time Software Testing and Debugging, *IEEE Transactions on Software Engineering*, Vol. 16, No. 8, August 1990, pp 897-916.

[VERV91] Vervoort, W. A., te West, R., Schoute, A. L., and Hofstede, J., Distributed Time-Management in Transputer Networks, *Proceedings of Euromicro '91 Workshop on Real-time Systems*, Paris, June 1991, pp 224-230.

[WILK89] Wilkenloh, C. J., Ramachandran, U., Menon, S., LeBlanc, R. J., Khalidi, M. Y. A., Hutto, P. W., Dasgupta, P., Chen, R. C., Bernabéu, J. M., Appelbe, W. F., and Ahamad, M., The Clouds Experience: Building an Object-Based Distributed Operating System, *USENIX Workshop on Experiences with Distributed and Multiprocessor Systems*, USENIX Association, Berkeley Ca, 1989, pp 333-347.

[WITT89] Wittie, L. D., Debugging Distributed C Programs by Real Time Replay, *SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, Madison Wi, May, 1988, SIGPLAN Notices, Vol. 24, No. 1, January 1989, pp 57-67.

[WYBR88] Wybranietz, D., and Haban, D., Monitoring and Performance Measuring Distributed Systems During Operation, *Proceedings of the 1988 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Sante Fe, New Mexico, May 1988, pp 197-206.

[YANG88] Yang, C.-Q., and Miller, B. P., Critical Path Analysis for the Execution of Parallel and Distributed Programs, *Proceedings of the 8th International Conference on Distributed Computing Systems*, San Jose, Californai, June 1988, pp 366-375.

[YANG89] Yang, C.-Q., and Miller, B. P., Performance Measurement for Parallel and Distributed Programs: A Structured and Automatic Approach, *IEEE Transactions on Software Engineering*, Vol. 15, No. 12, December 1989, pp 1615-1629.